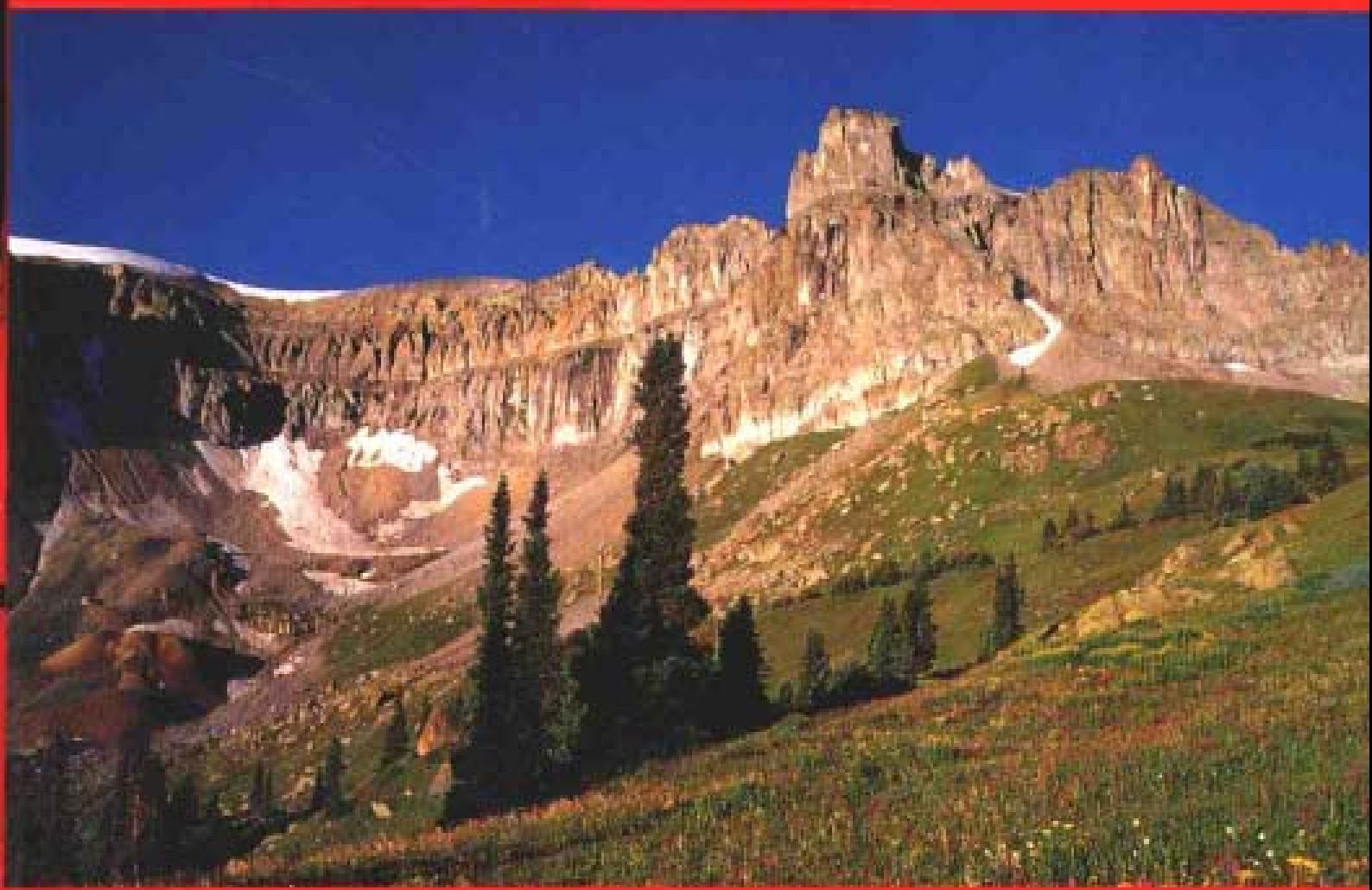
深入C++系列

Multi-Paradigm Design for C++

C++多范型设计

[美] James O. Coplien 著
鄢爱兰 周辉 等 译



Addison Wesley

中国电力出版社
www.infopower.com.cn

Multi-Paradigm Design for C++

C++多范型设计

C++是一种支持多种范型的编程语言：类、重载函数、模板、模块以及过程编程，等等。除了该语言的灵活性和丰富性以外，此前创建一种设计模式以支持在单个应用中使用多种范型的努力还很欠缺。

本书介绍了使用多范型设计的一连串框架，提供了形成正式多范型设计方法的基础的一系列超前的设计实践。

本书提供了利用C++的多范型能力的分析过程和设计过程的知识。书中使用易于理解的记法和易读的解释来帮助所有的C++程序员（不仅是系统构建工程师或设计者）在他们的应用开发中组合多种范型，以获得更加高效、健壮、更具可移植性和更容易复用的软件。

读者将获得对支持多范型设计的领域工程方法的理解。本书揭示了如何使用共同性和差异性的原则来分析应用领域，从而根据最适合每个领域的范型来定义子领域。多范型设计比任何一种技术或方法挖掘得更深，以处理软件抽象和设计的基本问题。

本书介绍了所有形成领域工程基础的概念和技术。这些概念包括：深入了解共同性和差异性分析，领域工程如何与常用设计模式相互影响，如何找到应用领域中的抽象，以及领域工程的原则如何用作对象范型的抽象技术的基础。最重要的是，本书讨论了在设计阶段如何将最适当的范型应用于实现的分析技术。

James O. Coplien 是对象范型和C++方面的主要专家和作者，自从进入AT&T以后，他就一直致力于对C++语言的研究。现在他是朗讯贝尔实验室的成员，他的著作主要集中在多范型开发方法和软件开发过程的人工组织学上。Coplien以前的著作包括：《Pattern Languages of Program Design》（与Douglas C. Schmidt合著）、《Pattern Languages of Program Design》第二卷（与John M. Vlissides和Norman L. Kerth合著）、《Advanced C++: Programming Styles and Idioms》（《Advanced C++中文版》已由中国电力出版社出版）。

ISBN 7-5083-1824-2

9 787508 318240 >

责任编辑：邹学三　曹娇
封面设计：王红柳

ISBN 7-5083-1824-2
定价：26.00元

深入 C++ 系列

Multi-Paradigm Desigu for C++

C++ 多范型设计

[美] James O. Coplien 著
鄢爱兰 周辉 等 译

中国电力出版社

Multi-Paradigm Design for C++ (ISBN 0-201-82467-1)

北京市版权局著作权合同登记号 图字：01-2002-4850 号

图书在版编目（CIP）数据

C++多范型设计 /（美）考帕里安编；鄢爱兰等译. 北京：中国电力出版社，2003

（深入 C++系列）

ISBN 7-5083-1824-2

Ⅰ.C... Ⅱ.①考...②鄢... Ⅲ.C 语言—程序设计 Ⅳ.TP312

中国版本图书馆 CIP 数据核字（2003）第 108179 号

丛 书 名：深入 C++系列

书　　名：C++多范型设计

编　　著：（美）James O. Coplien

翻　　译：鄢爱兰　周辉 等

责任编辑：郭学三　曾妍

出版发行：中国电力出版社

地址：北京市三里河路6号　　邮政编码：100044

电话：（010）88515918　　传　　真：（010）88518169

印　　刷：汇鑫印务有限公司

开　　本：787×1092　1/16　　印　　张：13.5　　字　　数：252 千字

书　　号：ISBN 7-5083-1824-2

版　　次：2004 年 2 月北京第 1 版　　2004 年 2 月第 1 次印刷

定　　价：26.00 元

译 者 序

C++是一种支持多范型的编程语言。C++之父 Bjarne Stroustrup 认为，“多范型”可以有一种比较朴实的说法——“多种编程风格”。多范型设计的两个主要原则是共同性和差异性。本书正是根据共同性和差异性的基础来确定“范型”概念的。多范型程序设计是综合应用多种程序设计风格的程序设计方法，是产生架构的活动的集合，其目标是找到能够最自然地表达应用领域结构的方案领域结构。多范型设计是一种技巧，它并不完全是一种严格的规范。

本书详细地介绍了从“应用领域”到“方案领域”的 C++设计实现方法，以及开发者在设计思考和设计实践过程中需要用到的记法、图表和设计模型。在读完这本书以后，读者将会了解如何根据应用领域的共同性和差异性分析来确定 C++方案领域的结构，也就是如何选择适用于应用领域同时又为 C++所支持的范型来形成方案领域的结构。

全书涵盖了这样一些重要的概念或方法，需要读者重点把握：软件族、共同性、积极或消极差异性、领域分析、领域划分、领域词汇表、面向对象的分析、复用、迭代、绑定、依赖关系图、范型混合，以及模式等。

本书第 1 章分析多范型设计的必要性，第 2 章和第 3 章分别介绍共同性和差异性分析。第 4 章解释如何使用领域分析来找到应用领域中的抽象。第 5 章说明怎样将领域工程的原则用作对象范型的抽象技术的基础。第 6 章应用“分析”来描述“方案领域”的特征，并且将 C++结构放到形成共同性和差异性基础的正式框架中。第 7 章综合考虑前几章的内容，介绍了结构复杂性不同的设计问题的分类，以及可作为基于领域分析和多范型设计技术基础的高级活动集合。第 8 章研究结构复杂的设计，并介绍如何组合各种范型。第 9 章补充说明了流行的设计模式与领域工程之间的关系，提供了对模式、领域工程以及二者关系的新的认识。

阅读本书需要读者一定程度地掌握了 C++编程语言，并具有一定的面向对象编程的经验。书中大量引用了相关著作，读者朋友可根据需要参阅参考文献中相关书目。

经过两个月的艰苦努力，本书的译稿终于完成并交付出版。译者由衷地希望读者朋友们能从这本译作中获取新的知识，提高自己的编程能力。本书由鄢爱兰、周辉主译，李明对全书进行了审校。欧阳宇、谢君英、谢俊、盛海燕、谢小花、李常青为本书的翻译做了很多工作。另外，吴晓晶、林知原也为本书的翻译工作提供了很大帮助，盛海燕和谢小花还完成了本书的初排和代码的录入工作，在此对他们表示衷心的感谢。由于译者水平和时间所限，译文中难免出现错误和不当之处，敬请广大读者批评指正。

译　者

2003 年 7 月

前　言

我很少像这次一样，花如此多的精力来为一本书命名。随着对原稿的修改，书名逐渐从强调一组基本元素（领域、工程、多范型、分析、设计、编程，以及 C++）变为总是强调一个或两个概念。出版社担心，使用大家所不熟悉的术语［“领域工程”（domain engineering）］难以迎合目标市场。审阅者 Tim Budd 担心他使用的“多范型”（multi-paradigm）会与本书中所使用的“领域工程”相混淆。而我则关心类似“分析”（analysis）这样的术语，因为我希望本书能对普通程序员（本书就是要介绍他们想要了解的问题）有所帮助。Tim Budd 婉转地提出，我们的要求分歧大到足以容纳“多范型”定义的宽泛区间。我坚持使用一个强调程序员的作用、而不是强调方法论者作用的书名。结果我们愉快地达成了一致，也就是形成了现在的书名——“C++多范型设计”。

我向来不考虑包含“模式”（pattern）、“对象”（object）、CORBA、“组件”（component）和 Java 类似措辞的书名。多范型试图比任何技术和方法挖掘得更深，以阐明软件的抽象和设计的基本问题。什么是多范型？分析、设计和实现之间是什么关系？这些问题涉及到支撑编程的基本范型的抽象基础。

一个最基本的问题是——什么是抽象？抽象是软件设计的关键工具之一。对于管理计算机系统无限的、不断增长的复杂性来说，抽象是必需的。此问题的共同答案通常都与“对象”有些关系，因此这也反映了过去十年或二十年形成的、用于支持面向对象技术的文献和工具的总体。但这种回答忽略了程序员经常使用的“共同设计结构”，它们并非是面向对象的：模板、重载函数系列、模块、通用函数及其他一些内容。C++中这种用法非常普遍，尽管它并不是惟一的。

有些抽象原则是与所有的这些技术所共同拥有的。每项技术都是依据它们共同的属性（包括各个实体互不相同的规律性）进行分组抽象的不同方法。对有些技术来说，共同性捕获了系统经常出现的外部属性，这些属性与系统的领域有密切的关系。对其他一些技术来说，共同性有助于使分析在递归解决方案中为领域所揭示的隐式结构规则化。

多范型设计兼具上述两个方面的特性。例如，多范型设计技术（被称为面向对象的设计）将对象分组为“类”，它描述了这些对象的共同结构和行为的特性。这种技术将类分组为层次或图表，它们反映了结构和行为中的共同性，同时允许结构和实现某个给定行为的算法中有规则的差异存在。模板可以用共性和差异的不同特性进行描述。共同性和差异性提供了宽泛而又简单的抽象模型，比对象和类都要宽泛，足以处理大部分的设计和编程技术。

共同性和差异性对软件设计模型来说并非是什么新鲜的事物。Parna 关于软件族的概念[Parnas 1976]至少在二十年以前就产生了。族是共同性相关的软件元素的集合，族的单个成员依据各自的变化而不同。从软件族中形成的设计思想，经常可以在被广泛接受的编程语言中获得体现。这里有一些很好的例子：模块、类和对象，以及通用结构。Lai 和 Weiss 在专用语言环境方面的努力将这一思想发挥到了极限[Weiss 1999]。注重软件族的发现的所谓分析活动，与注重如何表达这些抽象的所谓编码活动，总是紧密融合在一起的。多范型设计显然认可了语言、设计、领域结构，及它们表达共性和差异的方式之间的这种紧密联系。

我们在“领域分析”（它是另一个拥有很长历史的领域[Neighbors 1980]）的活动中找到软件族。软件复用是领域分析的最初目标，此目标与软件族正好相符。多范型设计显然注重对于复用比较重要的问题。为帮助设计者思考能对预期市场的范围自适应的软件，多范型设计将共同性（假设它不变）与差异性（假设它确实发生变化）显式地分离开来。我们要努力进行领域分析，而不止于一般意义上的分析。我们要设计抽象族，而不是设计抽象。如果进展顺利，这种设计方法将使系统长期获得更简单的维护（假设我们对差异性预测得很好）和更有弹性的架构（每次进行修改时不必“挖出”系统的“根基”）。当然，多范型开发只是帮助支持复用的技术目标的工具。只有在组织问题、市场问题和软件经济学的更大型的环境中，才能产生高效的复用。

我们使用共同性和差异性这些基础来确定“范型”（paradigm）的概念。尽管范型广泛使用在当代的软件设计中，但它只是根据共性和差异属性组织系统抽象的一种方法。对象范型根据基于结构中共同性的抽象，以及系统和算法中的行为和差异来组织系统。模板范型是基于族成员的结构共同性的，它将变化因素显式地转化成模板参数。重载函数形成这样的族：族成员共享相同的名字和语义，族的每个成员依据各自的形参类型相区别。

C++是一种支持多范型（类、重载函数、模板、模块、普通的程序性编程等）的编程语言。这正如 C++的创造者 Bjarne Stroustrup 期望的那样。多数程序员使用了对象以外的 C++特性（尽管有些人过度滥用这些特性，而另有一些人则在应当使用其他语言特性提供的更自然的设计表达时，对设计强制套用了面向对象的模式）。John Barton 和 Lee

Nackman[Barton 1994]的强大的模板代码或许是对多范型设计的不错尝试。

尽管 Stroustrup 指明了 C++是一种多范型语言，但 C++中并没有创建一种与丰富的 C++特征相配的设计方法的认真尝试。C++提供了多范型编程的一个特别丰富而生动的例子，其他的编程语言也同样获得了多范型开发的机会。当前的设计风格与当前实践中反映出来的 C++特性的预期用法之间存在一个鸿沟。本书使用简单的记法和词汇表来帮助开发者将多种范型有启发性地组合在一起，为这个鸿沟架起一座桥梁。

1995 年 9 月，我在 DePaul 大学演讲期间，系主任 Helmut Epp 先生建议本书使用“meta-design”这个术语，因为本书首先关心的是确定适合领域（软件正是为该领域而开发）的设计技术。这种看法对本书所采用的方法很有用，实际上，它描述了多数开发者是如何开始设计的。开发者首先必须决定使用什么范型，然后可以使用范型的规则和工具划分适合它们使用的系统。领域不仅属于系统构架者和设计者，它还属于普通的程序员。

决定使用什么范型是一回事，使用工具表达给定范型的抽象是另外一回事。我们可以使用共性和差异的原则来分析应用领域，以将其分成多个子领域，每个子领域适合用某种具体的范型进行设计。开发阶段进行的这种划分通常称为“分析”。但是，因为它试图创建实现技术能够表达出的抽象，所以将它当作设计的初始阶段更好。并非所有的实现工具（编程语言和其他工具，比如应用程序生成器）都可以表达全部的范型。因此，进行领域分析，（而不仅仅是应用领域的分析，还包括解决领域的分析）是很重要的。多范型设计将领域分析变成一种很明确的活动。解决领域的分析是多范型设计的“meta-design”本质的另一个方面。

本书有诸多特点。本书不是一种综合的设计方法、软件开发的生命周期模型、或是准备进行设计的方法。多数优秀的新设计都与以前的设计有所不同。我们很少会面临一个全新的或惟一的软件问题。对新系统中的每个模块都使用本书的记法和技术既不合适，也浪费时间。但在新问题出现的时候，我们应当准备好去面对它们，发现其中的结构，并将对它的理解运用到设计和实现中。而且，多范型设计的记法和技术为文件设计（可以为面向对象技术增加其他范型）提供了统一的方式。

多范型设计是一种技巧，它既不完全是一门艺术，也不完全是一种严格的规范。本书介绍了支持开发者思考过程的记法、图表和设计模型。由于有了所有的这些记法，我们总想尝试使用它们。多范型设计是产生架构的活动的集合，架构与各个部分之间的关系有关，但架构也与效用和审美有关——没有使用通常方法的优秀软件的属性。要获得良好的体验，有一个关键的地方，我并不期望多范型技术能够推动自动设计和编码。良好的体验部分来自经验，部分来自于良好的洞察力。因此，这也不是一本入门书。读者

应当具有一年或两年在大型系统中使用C++进行面向对象（至少）编程的经验。

把这些技术与常识一起使用，将很好地补充人们的判断力和经验。如果读者发现应用这些技术得到了一个既不喜欢又无法理解的设计，那么请不要采用这个设计。多范型设计技术只是工具，而不是指令。但所有的读者都应当通过本书认识到这样一点：对象范型或其他范型只是一组有用的范型，设计所表达的结构必须比任何单个范型所能表达的结构更宽泛。

本书的结构安排

为建立新的概念，并增加读者对领域工程和多范型技术的理解，每一章都建立在其前面章节的基础之上。多数读者都应按顺序阅读各章，然后返回具体章节作为参考。

第1章到第7章是基础章节，为领域工程打下了基础。

- 第1章介绍词汇表，分析多范型设计的必要性，并为领域工程打下高级的基础。
- 第2章和第3章分别介绍共同性和差异性分析。这两个概念在应用中是一起使用的，但为了介绍方便并强调其语言的精妙，本书中将二者分开讲述。第3章介绍了积极和消极差异性的主要概念。
- 第4章解释如何使用领域分析来找到应用领域中的抽象，这需要建立在前面章节所介绍的方法的基础上。
- 第5章说明领域工程的原则是怎样被用作对象范型的抽象技术的基础的。
- 第6章是重要的一章，因为本章以不同寻常的方式应用“分析”来描述“方案领域”的特征，并将C++结构放到形成共同性和差异性基础的正式框架中。
- 第7章将前面所有章节连接成思考设计的一个内聚框架中。它介绍了拥有不同的结构复杂性的设计问题的分类。同时，介绍了用以指导好的设计、并可作为基于领域分析和多范型设计技术基础的高级活动集合。本章论及这样一种简单的情况：每个领域都可用单个范型，并在很大程度上独立于其他领域进行开发。
- 第8章更进一步，研究了结构复杂的设计，简单的分而治之的方法对此并不奏效。本章为这种递归的、结构复杂的设计提供了多个示例，并对“打破递归”进行了启发性的介绍。
- 第9章补充说明了流行的设计模式与领域工程之间的关系。这种对比将会引起很多读者的兴趣，本章还提供了对模式、领域工程以及二者关系的新的认识。尽管本章并不是理解领域工程的核心内容，但它对理解用对象和其他范型（使用或者不使用领域工程）将多种模式结合在一起的当代设计实践非常重要。

本书中的领域名使用这样的字体：**Text Buffer**。模式名称也使用了这样的字体：**Template Method**。成员函数、类名和代码示例使用这样的字体：`Code Example`。

本书中的分类图表遵循统一建模语言（UML）符号。

致谢

非常感谢所有这些通过对话、评论和反馈提高了原稿质量的朋友。Van Den Broecke、Frank Buschmann、Paul Chisholm、Russell Corfman、David Cuka、Cay Horstman、Andrew Klein、Andrew Koenig、Stan Lippman、Tom Lyons、Lee Nackman、Brett Schuchert、Larry Schutte、Herb Sutter、Steve Vinoski 及 David Weiss 都在不同层次（从高层次的总体设计到最小的 C++细节）为本书提供了评论意见。非常感谢他们的帮助。我还要感谢孜孜不倦、充满耐心的编辑 Debbie Lafferty，她还参与了我的第一本书的编辑工作。与她一起工作真是一种乐趣。还要特别感谢我的制作编辑（production editor）Jacquelyn Young；感谢本书的审稿 Laura Michaels；感谢本书的排版 Kim Arney。与 Lalita Jagadeesan 的探讨激发我创建了多个有用的例子。还要特别感谢采编 Tom Stone，感谢他早期给我的建议，以及很早就在 Addison-Wesley 中给予本书的热情支持。特别感谢 Andrew Klein 在 UML 图表上给予的帮助。最后，要特别感谢贝尔实验室的管理层和我的同事们，尤其是 David Weiss，感谢他们让我分享他们的著作，以及他们所提供的支持和鼓励。

目　录

第 1 章
简介：多范型的必要性

本章介绍软件开发中的多范型，并将领域工程和多范型设计与面向对象设计相关的前沿技术和新兴主题结合起来。本章还介绍了以下重要词汇：软件族（software family）、共同性（commonality）和差异性（variability）。

1.1 领域工程和多范型

当代的软件似乎将“设计”与“对象”同等看待。对象范型提供了一种新的强大工具，使我们获得许多应用领域的共同抽象。同时，面向对象的设计已经成为一种得到普遍应用的工具。为实现这些设计，C++在本书编写之际仍然是首选的语言。然而，C++并不是完全面向对象的，并且多数实现使用了 C++的非面向对象特性。这表明，多数的设计实际上有一个非面向对象的重要的构件。假定大多数的项目使用面向对象的设计方法，那么这些非面向对象的结构从何而来？

这些问题涉及到软件设计的核心。设计的首要目标是满足业务需求，这种需求通常由用户期望来支配。设计的成功与否要根据对所建系统要服务的业务或领域的充分了解来判定。然而设计还有其他目标。比如，设计的实现应当可以理解，并易于建立（按照爱因斯坦的解释，应当是尽可能易于建立，而不是更容易建立）。设计还要尽量建立可以随时间演变、能适应新市场和新应用的系统。

实际上，所有这些设计目标是同一个基本原理的不同结果。许多目标都与业务的结构和业务结构如何演变有关。我们对业务的了解增加了满足客户需求的可能性。这种了

解也是系统结构以及有助于对设计进行表达和构型的词汇表的基础。通过了解业务的广泛、稳定的方面，我们可以设计出能良好演进的结构。

领域工程（domain engineering）是一种软件设计规程，注重对业务（一个领域）的抽象，目的是实现设计和工件（artifact）的复用（reuse）。复用是一个成功的设计的良好体现之一，并且，良好的设计技术会带来良好的复用，也会随时间的推移而带来可扩展性和可维护性。

多范型设计包含对象范型的许多目标，但这些目标的作用范围不同。对象范型也注重可扩展性和重用性。实现这一目标的方法是：将设计的共同和稳定的部分与不同和不确定的部分分离开来。经过简化后我们发现，稳定不变的行为和数据都包含在基类（尤其是抽象基类）中，而变量都包含在派生类中。当设计能够按照行为和数据结构这条线分成共同（稳定）部分与差异（不确定）部分时，上面的这种方法将非常适用。但是还有其他许多有用的方法，可以区分哪些是共同部分，哪些是有差异的部分。例如，我们可能要在一个相同的数据结构中选择 `short` 或 `long`，此时 C++模板非常适用——没有涉及到面向对象的问题。函数重载、函数模板及 C++语言的其他特性表现出其他种类的共同性和变化，它们比对象范型更加宽广。领域工程覆盖了所有这些需要考虑的事项。

设计是解决问题过程中的结构方面，包括对问题的抽象、划分及系统建模等活动，从而使设计者能够理解所要解决的问题。通过使用业务经验法则、耦合原理、内聚和子类型化以及对对象的非典型、非正式的任务分配，面向对象的设计能帮助我们“发现对象”。在一个非完全面向对象的领域中，我们究竟该如何建立设计抽象和实现模块呢？首先，很重要的一点是要认识到我们要处理的不再是单范型，而是多范型，每个范型都有各自抽象、划分和建模的规则。其次，很有用的一点是要知道大部分的软件范型——当然是指 C++所支持的这些软件范型——的特性都可以用更加通用的设计（涉及共同性和差异性）来描述。我们称之为“领域分析”（domain analysis）。领域分析建立了业务的一种模型（多种模型之一）。我们可以用这种模型建立解决方案的结构。设计的第一个要点是了解该模型构件的共同性和差异性。

领域分析揭示了抽象和工件的分组（二者通过它们的共同性——或许是通过它们差异性的相似本质——结合在一起）。这些分组被称为“族”（family）。如果说面向对象的设计是一个“发现对象”的过程，那么领域分析则是“发现族”的过程。注意，面向对象的设计是发现族的一种特殊情况：类为对象族，类层次为类族。因为它们有共同的地方，所以将它们组合在一起。除了通过这种密切关系，我们还可以通过对象范型使用“准则”（criteria）来发现其他重要的非面向对象的族——这就是多范型设计的重要性所在。

设计的第二个焦点是如何将这些共同性和差异性匹配到实现的技术结构（比如类、

函数、模板、类层次及数据结构）中。这称为“应用工程”（application engineering），也是多数人一听到“设计”这个词就会联想到的内容。如果我们理解了具体的范型所表示的共同性和差异性的种类，那么应用工程就可以使用领域分析的结果，为系统的架构和实现选择适当的范型。

本书展示了能够支持此类设计过程的形式记法、概念和简单表示法的一个框架。书中广泛介绍了基于领域分析的设计，以及该领域设计的一个实现。我将这种方法称为“多范型设计”（multi-paradigm design），它是针对领域工程（建立在C++编程语言所支持的一个小而丰富的范型集合之上）的具体方法。此方法可以推广至其他的编程语言（尤其是具有丰富类型的编程语言）及其他实现技术。这里强调C++是因为对“焦点”（focus）和“实用性”（pragmatism）的兴趣。

总之，领域分析是区分软件族的一套方法，而应用工程是实现和管理软件族的一套方法。领域分析和应用工程结合起来形成了称之为“领域工程”（domain engineering）的规范。多范型设计是领域工程的一种形式，它同时为应用领域和方案领域提供领域分析。在多范型设计中，应用工程建立在已有的工具和范型之上。

多范型设计和领域工程重新回到抽象的第一个原则和拓展对象以外的解空间的设计上来。这种拓展对设计、架构和实现很有意义。下一节按照共同性和变化的原则，建立了关于这些基本概念的词汇表。掌握了这些基本概念，我们就能更直观地理解大部分多范型设计的原则。

1.2 设计、分析、领域、族：术语定义

按照软件工程的惯例，“设计”（design）是指在分析和实现之间所进行的活动。“分析”（analysis）是指得出用户需求和用户期望的活动。在被软件采用之前，这些术语包含更广阔的含义。多范型设计正是迎合了这些术语更广阔的含义。本节介绍多范型设计的关键字和领域工程的词汇表。

在领域工程中，分析不仅是指得出给定客户需求的应用分析。我们还必须确定工具箱中需要哪些工具，必须根据我们的安排考虑工具和技术的局限性和优点，必须使用它们让设计者和用户都能获得最大利益。例如，设计者何时需要考虑是使用 **Unix** 库还是 **MFC**（Microsoft Foundation Classe）？这看起来像是一个实现决策，然而它通常是市场需求中的一个因素。很显然，这是影响最终产品的架构和功能性的一个决策。此分析必须在我们将其结果转化成一个实现之前进行。分析当然应该先于任何传统设计定义中的“设计”（即从需求中得出实现结构的过程）。但是，正如设计可以在实现的后期继续进

行一样，分析也可以在设计活动的同时继续进行。正在进行的开发可以洞察工具、方法和范型的选择适当与否。

1.2.1 分析

在多范型设计中，我们要讨论关于“应用分析”和“方案分析”的内容。应用分析是问题空间的“传统”分析。方案分析借用了应用分析的抽象工具和形式化方法，并将它们应用到“方案”空间中。同时理解“开发等式”（development equation）的两端有助于我们对适当的工作选用适当的工具，从而提高成功的可能性，甚至在应用分析阶段，设计者就需要确定工程工具箱的内容。

通常，术语“分析”表示“理解问题”的意思。术语“面向对象的分析”渐渐形成通用方法，这表明对象范型的设计原则为整理问题提供了有效的工具。不幸的是，分析期间使用的任何设计范型都会影响实现中使用的范型，即使实际上其他某个范型可以提供更好的划分。解决这个两难问题的方法是，在分析中避免任何的设计偏见。然而，这又导致分析不能直接预见使用可用工具的简单实现。通过考虑分析期间的所有实现范型，多范型设计解决了这个两难的问题。更确切地讲，该分析考虑了抽象和划分（这两者是大部分软件设计范型的基础）的基本原则。我们并不是在整个分析期间都进行划分，那样需要一个提供了划分规则的范型。分析收集这些信息来支持对一个和多个范型的选择。

1.2.2 设计

“设计”是为给定问题提供解决方案结构的活动。设计以一个问题开始，以一种解决方案结束。这里，“问题”表示当前状态与期望状态之间的一种“失配”（mismatch），这是一个足够广泛的定义，包括了 bug 修复和（系统）增强。设计必须与实效相权衡——由业务和易控制性等现实考虑所强加给设计的约束。与建立设计以及与某些机械工程规范类似，软件中的解决方案——设计的成果——包括“架构”（architecture）和实现。

我们仍然将设计看作这样一种活动：它带着我们从需求陈述阶段进入到实现阶段，但我们扩大了它的作用范围。由于分析需要同时仔细审查应用和解决方案，所以从某种程度上讲，分析也是一种设计活动。因为在多范型设计看来，实现工具就好比编程语言的表达结构，所以它涉及许多实现的问题。设计和分析都无法摆脱这种模型。这种看法使从应用结构到解决方案结构的过渡更加平滑，从而有助于避免结构化技术中数据流图之后发生的“相移”（phase shift），同时避免了将解决方案结构与问题结构匹配的错误预期。这是假定分析类直接映射到 C++类的早期面向对象方法的通病。

1.2.3 架构

架构是设计的主要产物。它是系统中有关事物的连接方式以及这些事物之间的关系。“事物”可以是对象、元组、过程及设计者的词汇表和工具箱中重点塑造的其他可被单独理解的“块”（chunk）。例如，在一个有限状态机（finite-state machine，FSM）中，这些“事物”包括状态、转换、转换动作、消息和机器本身。由于这些“部分”可能根本没有进行抽象，而是相当具体和完整的，所以我们用与上下文无关的词“事物”，而不用“抽象”。这些“事物”可能是逻辑架构的工件，它们在源程序中或者在内存中缺乏物理连续性。“过程”（process）就是这样的一个例子，它包含分散在整个程序中的子程序。

我们通常将非正式术语“结构”（structure）和架构关联起来。系统结构来源于我们放入其中的构建模块和装配这些构建模块的方式。构建模块的选择和安排紧接在问题陈述的需求与约束后面。架构帮助我们理解设计过程如何处理问题。多范型设计帮助我们选择适合可用解决方案结构的构建模块，并指导我们将这些构建模块综合起来，形成一个完整的系统。

通常，一个架构对应于一个“领域”，也就是为之建立系统的“关注”区域（感兴趣的区域）。领域可以是分层的，这就意味着架构模型也可以是分层的。

1.2.4 领域

领域是一个专门的或关注的区域。我们讨论的是应用领域——用户感兴趣的知识体。因为用户感兴趣的是应用领域，因此我们也将关注应用领域。我们将应用领域分解为多个“应用子领域”——分而治之。我们讨论“方案领域”，这是实现者主要关注的内容，但系统用户对此兴趣有限。任何给定的设计可能同时要应付多个方案领域，例如 C++结构、模式，也许还包括状态机和解析器生成机（parser-generator）。

1.2.5 族和共同性分析

软件设计中大部分的进展，当然是指那些在编程语言中得到使用的进展，都与形成和使用抽象的新方法相关。抽象可以不用涉及任何特定实例的细节来处理一般用例。当进行抽象的思考时，“我们关注哪些是共同的，同时抑制细节”。一个好的软件抽象要求我们充分地理解问题，以了解我们所关注的有关项之间有什么共同点，并且掌握是什么样的细节导致了这些项之间的不同。我们关注的项被统称为一个“族”（family），这些族——非单个的应用——是架构和设计的作用范围。无论族成员是否为模块、类、函数、过程或类型，我们都可以使用共同性/差异性模型，该模型可以应用于任何范型。共同性

和差异性是大部分设计技术的中心内容。

领域常常会（但不总是）包含族。族是一个“事物”（比如对象、函数、类）集，因为这些“事物”具有共同的属性，所以我们将它们组合在一起。共同性分析是用于形成和详细描述族的活动，共同性着重在第2章中讲述。

例如，现在有许多可在业务应用中使用的排序算法。在设计期间，设计者通常都忽略了这些算法之间的不同。即使所有排序算法的前置条件和后置条件相同也没有什么关系，我们不需要关心算法中的步骤。某个具体的程序可能使用多种不同的排序算法，每种算法都需要进行调节，以获得不同时间或空间的折衷方案。

如果我们使用某个抽象（比如“堆栈”）也是同样的道理，堆栈可以通过无限数目的数据结构和算法来实现。在进行设计时，设计者只关心堆栈要支持给定应用的族所需的操作，并要（通常带有预见性）推迟这些操作的具体实现。给定的程序可能使用多个不同类型的堆栈——“整型堆栈”（`Stack<int>`）与“消息堆栈”（`Stack<Message>`）的实现可能大不相同，但所有的堆栈都具有“共同的行为”（common behavior）。

在低级的设计和实现中，设计者开始关心不同类型的堆栈之间的区别。编程语言应当提供一种便利的方法以获得共同性。但同时，编程语言必须足够灵活地表达出这些共同性。每种类型的堆栈都有惟一的数据结构和惟一的算法。例如，整型堆栈可以使用单个向量，而消息堆栈可以使用一个顶点单元（head cell）的链表，其中的每个顶点单元包含了一个指向“消息”实例的指针。堆栈彼此之间的内部结构不同，操纵这些结构的算法也不同。

有些领域并未形成族，它们仅仅是焦点或关注区域。假设实时传输处理系统的设计中有一个过载管理（Overload Management）领域。当系统的负荷超过其实时处理能力时，Overload Management 接管系统资源（比如内存和实际时间）的配置。设计者可以在系统的生命周期开始时建立一次过载策略。可能会有一种过载设计可以不变地适应整个生产线。针对过载的代码可能在产品的整个生命期内都不需要更改（除了故障修复），但我们仍然想把它看作一个领域，将它本身看作一个值得重视的主题。

“好的”领域（可以容易地管理的领域）通常对应于子系统或系统模块。有些领域是跨模块的，在复杂的系统（有关复杂性的详细内容，请参阅第1.6节）中尤其如此。这里我们再以 **Overload Management** 作为一个领域的例子，此领域可能涉及到系统中许多部分的代码，这就意味着它干涉了其他领域。当这个领域发生变化时，其他领域可能需要对其进行跟踪。这种相互作用增加了设计的复杂程度。我们将在第 8 章对这些相互作用进行介绍。

1.2.6 抽象的“维度”（dimension）

软件设计的空间是多维的，具有设计的多个维（度）或“轴”：流程轴、数据结构轴、适用行为轴以及其他轴。每种设计技术都选择自身偏好的一些轴（共同性和差异性），并使用它们系统地阐述（定制）抽象。对象范型聚焦于具有共同接口的抽象数据类型的族，尽管这些抽象数据类型的内部实现可能不同。在多数范型中，多范型设计则更多地聚焦于共同性和差异性自身，而不是共同性和差异性的简单结合。

多范型设计使用了设计的特征维度（称为“共同性维度”）。重要的共同性维度包括结构、行为和算法。共同性和差异性的具体结合对应于“共同性类别”，此内容将在第 2.3 节更正式地介绍并将在全书中详细阐述。

1.2.7 精确抽象

“抽象”和“通用”并不表示“含糊不清”。相反，好的抽象可以精确地进行描述。例如，根据具体的职责（比如读、写字符序列）或可检验的属性（如果当前文件位置在索引 m 处，要成功读取 n 个字符，文件必须大于 m+n 个字符），我们可以定义一个抽象“文件”（File）。尽管这种描述在感觉上很抽象，只描述了可适应许多实现的行为和通用性，但它并非模糊不清。我们略过了不是所有情况都共有的细节，而保留了那些体现抽象的细节。

1.2.8 实现和工程

实现是设计继架构之后的第二个产物。“实现”（implementation）表示实际代码——不仅是图表或形式化的表示。许多当代的方法都将设计看作是架构和编码的中间阶段，而不是将架构和编码看作是设计的产物。但从更加实际的观点来看，我们不能将设计与其他架构或实现分离开来。如果设计是给出解决方案结构的活动，而架构是针对结构的，那么“设计”不正是表达产生设计的活动的恰当的术语吗？大量的代码也是针对结构的。为什么不也用“设计”来表示？如果了解程序员是如何工作的，大家将会发现：在许多应用领域中，他们并不真正描绘出架构、设计和实现，而不管正式的自主方法是否要求这样。（在检查设计文档之前，编码工作已经完成了多少次？）通过对类进行编码，面向对象的设计者深入理解了对类的职责的分配。对软件设计流程的经验研究表明：在进行设计检查时，多数开发者至少有部分编码的解决方案，并且，设计决策将持续到编码的最后一刻[Cain+1996]。

多范型设计还尽力找出实现的约束和机会，而不是将实现看作是设计之后的内容而

留给“实现者”。如果设计需要产生一个解决方案，此方案必须是可用的。这就是设计的“工程”（engineering）部分发挥作用的地方。工程依靠已证实的技术、科学的原则以及问题解决者的创造性思维解决一个问题或一类问题。工程的进行向来都着眼于实现，这是非常实际的。领域工程作为一个整体，将问题的解决从注重单个问题的活动提高到解决问题族的活动上。也就是说，领域工程通常会重现设计的问题。多范型设计——领域工程的一种具体形式——统一了抽象技术和整个开发过程所使用的词汇。多范型设计的应用工程构件无需延至开发过程后期。领域工程的形式记法也是领域分析活动的重要构件。

1.3 超越对象

许多读者可能会问：“我已经使用了面向对象的设计技术，尽管偶尔会有一些问题，但事情还是以某种方式趋向于得到解决。为什么我要将这些技术放在一边不用，而去考虑多范型设计？毕竟，C++是一种面向对象的语言，不是吗？”

这种反对的理由的一个潜在危险就是：术语“面向对象”已经变成了“好”的代名词。多范型设计提供了可用于描述对象范型的形式记法，比临时使用的记法更加精确。而且，它证明了这些形式记法与诸如去耦合和聚合、抽象以及演化等重要软件目标的关联。

如今的市场上，每种可能的范型都贴上了“对象”标签。由此产生了混合的设计环境，它是对以前的设计环境进行附加来建立的，通常为一种期望所支持，这种期望依赖于对旧的方法和技术的投资（人力、工具、嵌入式软件甚至是信誉）。这些环境中的大多数被称为“面向对象”的。这种组合方法的一种倾向是：它们模糊了面向对象设计的一些中心原则，用其他方法中的原则替代了它们的位置。设计技术的这些混乱的组合会导致架构性的灾难。一种常见的失效模式就是围绕系统数据结构使用结构化分析建立初始设计，然后使用职责驱动设计来实现。这种技术的共同结果是：原始数据模型的聚合被破坏了，越来越多的对象的内部数据结构被迫暴露。维护变得困难，整个系统变弱，维持系统运行需要花费大量精力。

但纯粹的对象也不能解决问题。优秀的范型曾经被面向对象的大肆宣传所排挤。当代的开发工作室中的情况是，没有人愿意总是使用过程分解，即使对于批量排序例程——对象也以某种形式用在解决方案中。如此产生的设计就如同强行把方钉子打入圆孔。

将多种范型恰当地混合在一起总不失为一个好主意，杰出的软件设计者总是在他们的工具中长期保留多种工具。Peter Wegner 曾经这样评论：对象范型本身是建立在它之前的范型基础上的“混合物”，包括模块化、抽象数据类型、过程和数据结构。C++还进一

步在平等地位上支持过程性、模块化、基于对象、面向对象和通用的编程。混合范型很难被接纳，就像很难将多所建筑学校顺利地混合在一栋楼里一样。很少有被广泛理解的形式记法、指导方针、方法或混合范型的规则，所以混合范型由特定的和政治的考虑所推动。在倾向于将 C 和面向对象的开发习惯混合起来的 C++编程项目中，这是一件尤其重要的问题。

本书激发了对多范型的需求，并为多范型开发的“半正式”处理方法铺垫基础。本书提供了词汇表和记法，可用作可复用设计方法的基础，也可以在逐个用例的基础上作为对已有方法（比如，面向对象的方法）的试探性补充。书中介绍了基于共同性和差异性的分析形式，以及已经成为多数设计方法的基础的直观概念。许多面向对象的设计本能地脱离了共同性和差异性分析。类层次是将软件族联系在一起的一种方法。对象和共同性分析之间的关系是第 5 章要讨论的主题。但也有许多族没有获得最优实现，或者没有被当作类层次来考虑。有些技术在领域工程和多范型设计之外的面向对象设计中表现优秀，这些技术适当地补充了多范型设计。

1.4 共同性和差异性分析

因为共同性和差异性分析是人脑形成抽象的本质，所以它们也就形成了设计的基础。好的抽象技术表达出了软件的复杂性，并帮助设计者得出并组织领域知识。“复杂性”和“问题定义”是当代实际软件开发所面临的两个主要挑战。我们将根据共性和差异来定义“范型”的概念。

共同性和差异性这两个简单的概念是本书中大部分原则的基础。每一章都将使用并详述共同性和差异性的基本知识。我们已经见过通过它们来确定和描述软件族的应用。它们还描述了主要的设计结构（第 2.1 节），它们所带来的结构将会进展得很顺利（第 2.6 节）。并且，它们可以帮助捕获、构造和编纂领域词汇表（第 4.3 节）。它们确实自始至终都是面向对象的分析的基础（第 5.1 节）。实际上，共同性和差异性分析是许多设计技术的有力推广。它们涵盖了 C++从过去二、三十年编程语言开发中所继承到的大部分抽象技术。

1.5 软件族

将类似的实体和概念组合在一起，这是我们进行抽象的一种方法。如果根据算法步骤的相互关系将它们组合在一起，我们称这种抽象为“过程”。如果将相关数据的封装集

合的相关职责组合在一起，我们称这种抽象为“类”。

我们可以根据许多流行的范型中所没有的标准进行分组。为了进行抽象，我们从可以发现共同性的地方找出它们。这些共同性有很多维度：结构、算法、语义、名字、输入和输出、绑定时间以及默认值——几乎都可以根据任何内容对它们进行分组。根据Weiss[Weiss1999]的FAST（Family-oriented Abstraction, Specification, and Translation，面向族的抽象、规范和翻译）分类方法，本书中我们将这些分组称为“分组族”。软件族的概念更早源于Dijkstra[Dijkstra1968]和Parnas[Parnas1976；1978]。Parnas将族描述为通过共同性紧密联系成一组的条目，共同性比组成员之间的差异更重要[Parnas1976]。

Wesis详细描述并扩展了Parnas的定义。族只是系统部件（甚至概念，比如过程这样的非连接部件）或抽象的集合，因为它们的共同点多于不同点，所以将它们归为一组来对待。因为族成员有如此多的共同性，所以通常可以将它们看成是相同的。我们明确地描述出它们的不同（称为“差异性”），以区别各个族成员。族的这些特征描述将继承层次结构，带入到C++专业人员的头脑当中。但对于模板、重载函数和多数C++用来表达设计结构的其他语言特性，它也是同样有效的特征描述。

共同性和差异性形成了族。我们可以与生物学上的族（它们的遗传构成了共同基础和显著特征）进行类比。根据生物的遗传共同性来定义族，并根据遗传的差异来区分族成员。Wesis认为：我们可以将差异性当作基于设计期间所决定的顺序而生成的“族树”（family tree）。

程序的族由族的部件构成。用微分方程来解决程序问题很可能会包含微分方程包（算法族）的族。一个无线通信系统将包含针对不同类型的线路和信息通路的软件族。编译器可能有针对不同优化类型、针对不同目标平台以及支持某个调试功能领域的代码生成器。

很多软件族由现存的编程语言结构自然地表达出来。在一个继承层次结构中的类组成族，族的根部为源祖类，枝叶部分为变异类。它们的共同行为（假设一个子类型的层次）是基本的基因集合。每个类可以改变它从父类继承到的属性。我们可以有一个基类`LogFile`，三个派生类`RemoteLogFile`、`UmmarizingLogFile`和`ScavengingLogFile`。

但有些族分组不适合用面向对象和面向过程的结构来表达。有很多计算机科学结构可以表达共同性和差异性，但它们却不是编程语言结构。数据库词汇表拥有大量的这种表达方法。过程、任务和线程形成了操作系统平台上族的类别。网络协议是消息族。甚至在编程语言中，我们也发现了规则、任务、异常、模式和很多其他的分组。这些在日常的设计中也很重要。

共同性定义了跨应用抽象而不变的共享内容。一旦我们发现了共同性，它就无法再

引起我们的兴趣了。我们认同它并将其放在一边，不再深入讨论它。差异性获得了区别领域中的抽象的更多相关属性。在分析中揭示差异性，并提供实现的方法以表达出它们，这两点都很重要。在 FAST 中，面向应用的语言（在编译器及其环境中）中的共同性是隐式的，而差异性在语言结构中却被显式地表达。

我们可以直接类推至面向对象的编程语言，它们的程序通过签名中的共同性来对抽象进行分组。类层次形成了族，新的类只需要捕获它们是如何不同于其他族成员的。在使用这些类中的任意一个时，都可以认同它们的共同性，并可交换使用它们的实例。

1.6 多范型设计

多范型设计是一种领域分析技术，其特性是应用领域分析和方案领域分析同时进行。多范型设计的目标是找到能够最自然地表达应用领域结构的方案领域结构。我们要将方案领域的共同性和差异性与应用领域的共同性和差异性进行比较，以理解什么样的解决方案技术被应用到问题的哪一个部分。为了支持这一活动，多范型设计提供了建立方案领域结构的分类法的技术。分类法描述了各种共同性和差异性，它们与每一种范型以及表达这些范型的 C++语言特性有密切的联系。设计者可以使用这些紧密的联系来为应用领域分析所产生的共同性和差异性选取适当的解决方案结构。

1.6.1 语言：C++

本书“多范型”中的范型是用 C++编程语言描述的范型。这是范型的一般用法——“范型”这个字眼通常暗含函数的、数据库的和基于规则的范型，这远超过了 C++的范围。推广到 C++或其他任何编程语言所支持的这些结构以外的正式结构，“范型”仍然是一个适用的术语和技术。范型被推广（也称“泛化”）到对象范型（它本身经常被看作比其他技术更通用）之外。第 6 章描述 C++方案领域的分类法，是对 C++本身进行领域分析的结果。

多范型设计帮助设计者使应用领域共同性和编程语言提供的共同性紧密合作。本书中，我们将着重讲解作为这样一种编程语言的 C++。对象范型恰恰表达了一个熟练的 C++程序员可以使用的多种共同性和差异性中的一种。多范型设计总是试图显示编程语言的其他特性（只要这样做是有意义的），以便于程序能够更好地表达一个好的设计划分。

尽管多范型设计原则与其他方案领域的工具有紧密的联系，但本书不直接介绍超出 C++范围本身的多范型。数据库和调度（scheduling）抽象——不可否认，二者都很重要——不在本书介绍的范围之内。《Introduction to Database Systems》[Date1986]是一本经典的

数据库方面的优秀书籍，同时在《Object-Orient System Development》[DeChampeaux+1993]一书中可以找到关于并行操作的出色的基础知识。

1.6.2 处理复杂的族

有一点很重要，就是要认识到软件族并不是应用实体的分离集合或子集的简单描述。与生物学一样，族是相互交迭的。就像族之间的通婚以及混合了来自多个父母的遗传特性所产生的族成员一样，我们在软件架构中发现了丰富而复杂的关联。任何复杂的系统都有许多相关的划分，许多复杂系统都可以使用多重、同步而又独立的视图获得最深入的理解。每个视图代表了特殊观察者的一组特殊关系：性能、资源使用、信息流等。例如，一个无线通信系统可以有用于记账、独立适销特性（比如呼叫等待）、诊断、初始化、容错、审查和许多其他方面的软件族。这些软件族很少会形成代码的分离子集。相反，它们相互交织形成一个多范型的系统。

族或者系统的视图必须进行细致地选择、混合和管理。容纳多样、同步视图是比较困难的，但有了多样的视图，系统架构的重要语义常常被忽略了，这是多数软件设计技术和实践的重大缺陷之一。早期的面向对象设计常常忽视了用例的重要性，这些设计最终在 Jacobson 等人的作品中给出[Jacobson +1992]。看起来好像每种范型都通过贬低其他的范型来使自己的划分标准得到注意，这是为了避免在单个工程中包含多种范型而采取的市场和政治活动。复杂系统中能够混合多种范型，这一点尤其重要。复杂性与区别的数量成比例，同时也是系统的富有意义的视图。也就是说，复杂性与系统中族的交迭数量成比例。在没有使基类——描述类层次作为一个整体的领域语义的特性——脱离任何领域的语义之前，我们不可能将任何复杂性系统简化成单个的类层次。这会在没有必要使用类对象作为共同基础的系统中表现出来。

在复杂性的下一个层次中，设计者可以将系统划分成类类别[Booch 1994]，它是核心部分带一个类层次的子系统。由于层次之间潜在的耦合，这并不理想。这种耦合常常来自于过程的抽象，与此同时，类层次捕获结构的抽象。例如，假设有这样一个设计，它有单独针对 GUI（Graphical User Interface，图象用户界面）和应用对象的层次，那么管理程序和用户间交互的代码应该放在哪里？它是一个过程因素，不完全属于一个层次或其他层次。这种代码可以放在其自身的类别中，但这么做将在层次之间创建一个复杂的耦合模式。像 MVC（model-view-controller，模型-视图-控制器）这样的模式为系统视图提供最好的连续性（standing），却没有为领域创建一个单独的、物理清晰的层次。MVC 捕获了领域之间的依赖关系。这些依赖关系不总是表现为模式，我们需要更通用的设计机制来处理领域之间的相互作用。

对象共同性就是用来处理这个问题的。随着对象规范的成熟，呈现出越来越丰富的机制来解决这一问题：20 世纪 80 年代早期的多重层次、20 世纪 90 年代类型和类的分离、20 世纪 90 年代中后期基于任务模型的普及，以及同一时期的设计模式。从 Cartesian 最初在面向对象编程的基础上“分而制之”的继承模型到现在，我们已经走过很长的一段路。

鉴于多数领域工程技术将领域和模块、子系统、库或其他自主式结构同等看待，随意多范型设计也支持多个领域的相互交织。多范型设计用于处理结构、名字和语义（包括输入和输出的语义）、行为、算法、汇集时间、粒性以及状态所描述的族。多数的方法都建立在这些原则之上，尽管它们使用了一些高级的正式结构概念（比如对象和模块），而不是使用更一般的（低级的）多范型设计概念。多范型设计根据这些更一般的概念描述了领域的交互作用。

1.6.3 合成模式

第 9 章研究软件模式和多范型设计之间的关系。设计者可以将结构模式与多范型设计技术结合在一起，第 9 章将论证：许多结构模式（比如 Template Method 和 Strategy [Gamma1995]）只是多范型设计技术的特例。其他的模式不在多范型设计的范围之内，并提供了不易于用一种方法或“半正式”的方法（比如多范型设计）获得的设计手段。这些模式是多范型设计的补充。

1.7 多范型开发和编程语言

许多分析技术只注重应用领域或“问题”领域。通过远离解决方案的细节，这些方法声称自己更加抽象，并获得了跨多种编程语言的应用性。例如，有些方法论者声称：在对象范型下进行分析甚至是设计，并用所选择的任何技术来实现都是可能的。虽然使用这种方法勉强可以在一个“新建应用（greenfield）”系统中成功，但为长期的维护留下了很多问题。

语言指导、约束、支持并表达了我们如何划分一个应用，以及如何操作它并赋予其形式。举个一般的例子，Smalltalk 中的同类 List 在代码上具有一个封闭的的形式，以服务于所包含元素的所有可能的类型，从而在运行时处理所包含元素的不同类型。在 C++ 中同样的实现可能会使用模板为每个族成员生成其代码的拷贝。这里，实现技术必须考虑进去。

多范型设计在应用领域和方案领域进行共同性和差异性研究。我们同时在这两个领域中使用共同性和差异性分析，并调整这两种分析以获得解决方案的结构。这与 FAST

方法略有不同，后者是从领域分析中获得实现技术的。

1.7.1 FAST中的面向应用的语言

Weiss等人[Campbell+1990]最初的FAST方法很明智地考察了应用领域和方案领域，以及二者的相互关系。好的分析的关键是收集与共同性有关的设计知识，并将其作为“设计秘密”隐藏起来，从而使它不至于弄乱了开发者用于表达设计的代码。从理论上说，编程语言自身应当隐藏其所表达的领域的全部共同性，以便程序员可以将精力集中在表达差异性上。

例如，电子表格语言隐藏了自动更新值单元的共同机制。这是多数单元和不随应用发生改变的单元的共同的逻辑。但是，多数电子表格语言提供了丰富的结构来表示随应用变化的值和结构。

在最初的FAST方法中，Weiss使用共同性和差异性分析来描述软件工件和抽象的族的特征。FAST建立了族成员的一个结构化词汇表，作为针对每个领域的自定义编程语言的基础，而不是随机选择编程语言。这种语言表达了族成员中的差异，这里假定族共同性包括了所分析领域全部的抽象。针对从规范中产生族成员的语言存在一个翻译器，由此可以利用共享的共同性。这种语言以及其相关的支持工具为领域形成了一个应用工程环境。FAST方法是Campbell、Faulk和Weiss在合成过程[Campbell+1990]上早期成果的改进。它的重点在通常使用面向应用的语言（application-oriented language，AOLs）的应用工程环境上，AOLs也称为little language[Bentley 1988]或专用语言。

例如，可以使用FAST技术来分析一个协议，然后使用分析的结果来指导AOL（简洁地表达协议的应用领域中的解决方案）的创建。FAST技术对应用（问题）领域进行分析，以作为方案领域的结构约束的来源。很少有来自于方案领域自身的约束事项。如果要复用方案领域的技术，而不是为每个领域都定制一个新的应用工程环境，方案领域约束就变得很重要。

有些领域从AOLs中受益匪浅，进而调整了所定制的应用工程环境的费用。AOLs使在领域中快速生成解决方案变得更简单。有些领域支持正式的分析（比如为解开协议领域例子中的死锁而进行的分析）和真正的优化（通用语言很难实现这一点）。当这些因素支配了工程的需求时，从长期来看，开发一个AOL和支持它的应用工程环境就是划算的。面对这种决策的工程能够从划算的分析（考虑了每一个领域）中获益。作为一项初步的工作，这种决策可以对每个领域独立地进行。

应用工程环境方法的一个不利因素是存在初始构造费用，以及工具、过程和支持人员的长期维护的费用，尤其是当这种费用不能跨多个工程广泛延伸时。另一个不利因素

是它缺乏对从分析到编程语言的翻译的引导。一种语言（比如 AOL，而非通用语言）的成功要求设计者对应用领域和语言设计都非常熟悉。即使对于一个优秀的语言设计者，好的语言也非常难以设计，而且好的语言设计可能（或应该）需要几个月的时间。这个问题可以通过进行全部共同性和差异性的面向对象的分析来处理，以发现方案领域语言的线索，正如 Weiss 和他的同事在没有发表的作品中所做的那样。

多范型设计从任意单个范型后退了一步，以推广这个过程。而且，它的重点在尽可能地复用已存在的方案领域结构。这就是为何要在多范型设计的分析活动同时考虑问题领域和方案领域的原因所在。

1.7.2 领域分析和 C++编程语言

Weiss 的原则是本书的大部分技术规则的基础。我们只在选择编程语言时将公司分开。多范型设计并不是从应用领域分析中产生常规编程语言的结构。相反，多范型设计是从一个丰富的、通用的编程语言（比如 C++）中提取出结构，并让此结构满足应用领域的需要。多范型设计使用这种结构来指导设计，除了比单独的 FAST 过程更具灵活性之外，而且比最初的综合过程更具规律性。

共同性和差异性分析的技术比任何编程语言都要宽泛。我们本来可以选择任何不依赖于习惯而直接表达反复出现的设计结构的语言的。高级或具有创新精神的设计者能够通过使第 6 章中开发的模型模块化，并将它们应用到本书其余的分析中去，从而将本书的技术改编成这样一种编程语言。实际上，本书中的许多素材几乎都与 C++或与编程语言无关，尤其是前面的几章。

那么为什么要讨论 C++？毫无疑问，C++的普及是本书的技术如何能被广泛应用的一个因素。这是与应用工程环境中的 little language（几乎总是为小型的本地客户服务，并享受本地向导的支持）有关的重要事项。但讨论 C++更重要的原因是它能很好地支持多种范型。C++有模板、重载函数、继承、以及虚函数——一个丰富的设备集，可以表达共同性和差异性模式的宽阔领域。在 C++中，我们可以编写单个 List 模板（一个源程序集合）以保存多种类型对象的列表。在 Smalltalk 中我们也可以做同样的事情，尽管使同样的类成为可能的机制是以非常不同的方式实现的。这里先不说其他的语言，C++的一个优点就是它直接表达了分析的意图，一般来讲这种方式只有 Smalltalk 才能使用。

FAST 方法的应用工程组件超出了任何单个通用语言的共同性和差异性。它的应用工程推荐使用可以捕获特定领域的共同性和表达特定领域的差异性的专门语言——此语言帮助开发者聚焦在不同的属性上。背景共同性提供了一种语境，程序员可以依靠它在整个领域中始终保持正确的认识。FAST 促进了需要简单、富于表现的语言和环境的开发文

化的形成。如前所述，要得到一种好的语言和环境需要付出相当的代价，而且它们只在表达某些领域时才表现得不错。C++是一种真正的通用语言，它善于处理 little language 无法表达的、受到广泛支持的领域的复杂性。如果不考虑 C++的其他复杂性和倾向，通用语言的这些优点有时超过了 AOLs。

鼓励我们在 C++中使用多种范型的原则同时也鼓励我们在更高层次上使用多种设计策略。有些领域非常丰富而繁杂，使用 little language 和 C++的结合方式可能是最佳的解决方案。我们可以使用多范型开发来构造一个 C++类库，用作 AOL 的基础。另一种方法是使用 C++对一个领域进行划分，然后用 C++来处理某些子领域，用小型语言来处理其他子领域，这两部分将通过一个架构接口进行通信。编译器的编写者们经常这样使用 yacc。常识和经验通常会为我们提示一种或多种这样的方法。

1.7.3 多态性

多态性是 C++所支持的这些范型中出现最迟的。“多态性”（polymorphism）通过对象范型进入了主流技术术语，就对象范型而言，使用这个术语通常意味着运行时间操作符的查找。虚函数支持 C++多态性的这种语义，而重载、模板、继承、甚至类型转换只是多态性的形式。我们将在第 6.9 节深入讨论这一点。“多态性”的字面意思为“许多形式”。如果某个事物有许多种形式，那么这就暗含了一些共同性，通过这些共同性就可以建立和判定这些形式的差异性。共同性建立了一个框架，在这个框架内多态性可以有明确的意义，而差异性则描述了多态性自身的特性。

例如，如果我们在一个绘制形状的图形包中使用多态性，那么差异性就是不同类型形状的绘制算法和数据结构。但所有的形状都有一些共同点：它们共享某些数据结构（比如它们的位置、边界和填充色），并且它们都共享某些行为（都可以移动、重绘、创建和删除）。共同性使它们成为“形状”，而差异性使它们成为“不同的形状”。多态性利用了共同性和差异性的这种平衡，其结果是一种描述了整个族的特性的抽象（一个抽象基类）。

当然，我们不必局限在面向对象的例子上。所有的集合都有共同的地方：它们能够将所有元素插入到集合本身，并且它们可以共享算法和数据结构。但“整数集合”与“窗口集合”是不同的。这是因为它们插入新元素的逻辑使用了不同的的算法，以判定被插入元素与集合中已经存在的元素是否相同（集合中不能存在相同的元素）。这也是多态性。多种形式的集合共享着很多的属性，但每种集合又在细微的地方有所不同。我们可以使用模板来捕获这些不同点，这里的模板是多态性的另一种形式。单个的抽象（一个模板）描述了整个族的特性。

1.8 共同性分析：其他方面

多范型设计使用了传统的计算机软件设计的有关技术。后面的章节中我们讨论其中的一些技术。这些类比会帮助我们将多范型设计与其他更广泛的设计模型联系在一起，从多个视角来检验多范型设计。这也帮助我们认识了为什么多范型设计的原则是可行的，以及为什么它们非常重要。

1.8.1 策略和机制

“策略”（Policy）和“机制”（Mechanism）的分离成为许多经久不衰的软件工程原则的基础，我们可以将多范型设计当作这些原则中的一员。对于第一级近似来说，市场并不关心策略制定以后用于实现策略的机制。在很多时候，机制要回答“如何实现功能性”的问题，而策略则提出了“要实现什么”的问题。

一个业务领域的机制通常是随时间保持稳定的，而策略则随着市场的需求、规章和其他外部影响而发生改变。因为机制是稳定的，而且是系统客户第二位感兴趣的（它们很少出现在需求中），所以我们要在用户无法看见它们的地方将“机制”隐藏起来。类接口之后的这种秘密隐藏的实现是对象范型的一个基本目标。当然，对对象来说，这种思想并不新鲜。它是模块设计的一个早期原则。最终用户不应当看见机制，但应当可以操控设计（对应于业务领域的策略）的这些部件。这是 Parnas 的信息隐藏的一种基本精神。在很多好的系统中，共同性捕获了机制，差异性捕获了策略。

在多范型设计中，通过使用所选择的编程语言（本书中为 C++），我们表达差异性（策略），并隐藏共同性（通常为机制），为每个应用领域构建一个框架，并将领域共同性隐藏在其中。框架有外部“挂钩”，程序员可以操控或扩展此“挂钩”，从而为某个具体的应用调整框架。这些“挂钩”可以表达成 C++的抽象基类（程序员提供一个派生类）、模板（程序员提供模板参数）和重载函数族（程序员可以扩展该族或调用已存在的函数）。它们也可以通过其他的语言特性表达出来。类似这样的通用方法保持了设计的灵活性。

1.8.2 随时间的差异性与随空间的差异性

在分析期间，我们可以采用两个不同的角度：聚焦在随时间的变化或聚焦在一个族内的差异。我们可以将族中的差异当作“空间”中的“事变”（occurring），当作在时间上共存的不同客户和市场的不同配置。这两种方法是分离的，但目标又是相关的。

通过聚焦某个产品或生产线的抽象族的共同性，我们提升了架构的抽象，减少了在开

发方面的努力——这是最大限度的复用。我们可以注意到：围绕着它们不同的参数，有几种不同的窗口系统技术和设计。这就允许我们部署这样一个系统族，该族的每一个成员结合了一种不同的窗口技术。这是最经常与术语“领域分析”相关联的策略。

通过聚焦什么将随时间保持不变这个问题，我们努力降低维护的费用，我们努力预测领域是如何变化的。如果情况更好，我们就努力预测领域是如何不发生变化的，将其映射为一个共同性，并在系统的结构中捕获它。差异性应表达什么是随时间而发生变化的。对一个好的设计的测试是随系统的生命周期预期市场变化，并预测它们对共同性设想的扰乱有多严重。如果预测错误，我们将为此付出昂贵的代价，并会导致这样一种情况——“所有的这些变化都是常见的”——在这种情况下，一个小的需求变化也需要很大的开发力度。历史通常是未来的最好预见者：在过去保持稳定的事物，在将来也通常会保持稳定。

空间的共同性常通过它们随时间的共同性表现出来。在时间上，系统设计者可以调整变化的参数，从而在连续的版本中逐个选择不同的族成员。项目中也可以引入开始设计时没有预见到的族成员。在一个好的设计中，因为共同性分析总是试图预测随时间或随空间的变化，所以这些“后来者”可以利用一个良好的共同性分析的预见。

与任何使用领域分析的过程类似，在多范型设计中，设计者必须对空间中的差异和随时间的变化进行细致地预测。通常时间测试是两种测试中的首选。我们可以从已经存在于成熟的领域中的应用来进行推广，这些推广指出了时间测试将会遇到的共同性（记住，过去预示着将来）。在新的领域中，及时规划时间上的共同性和差异性更加重要。

1.8.3 后期绑定

Gregor Kiczales 指出，要尽可能迟地“绑定”所有的决定，以便给予客户对系统进展的控制权[Kiczales1994]。推迟绑定是对象范型的一个重要部分，也是面向对象的程序员所熟悉的“多态性”的一个关键部分[①]。我们要使用编程语言的可用设备和其他实现技术，以支持变化的语言结构来调整领域差异性，同时争取最小的连锁反应。

预测总是很困难的，我们无法预见系统将要发生变化的所有方式。软件的提供者和用户有时必须改变系统的策略和系统的机制。在用户不需要与机制打交道的地方，我们使用 FAST 原则努力将它们隐藏起来。但是我们能使每一件事物都很容易被改变吗？

Kiczales 认为这就是为什么 meta-object 协议如此重要的原因：它们使程序员和系统之间合同的改写成为可能。C++没有直接演进其对象模型的设备，类似的功能性来自代码

① 有关多范型设计和多态性之间的关系，请参见 6.9 节。

模式[Coplien1992]和模式[Gamma1995]。

通过对共同性分析本身进行共同性分析，寻找对多个领域规则化的机会，我们可以获得更广泛的灵活性。根据 Kiczales 的提示（“如果不能征服，那么就分解，并尝试一个 meta-object 协议”），我们在方案领域获得的成功可以比在问题领域中多。当发现这个问题在符号编程语言中得到解决时，我们必须用前面提到的各种约定（详情请参见[Coplien1992]的第 8 章和第 9 章）在 C++中进行模拟映像。这在设计模式（第 9 章，用于编码那些不易于被封装到对象中的表示语义的反复结构）方面为我们带来了更多的收获。该模式提供了一个词汇表，以提升程序员和系统之间的合同等级，这与 Smalltalk 和 CLOS 的 meta-object 编程结构的方法有些类似。

值得注意的是，Kiczale 最初的一个很有发展前途的概念“方面”aspect[Kiczales1997]与用在领域工程中的概念“领域”密切对应（当然也与本书中的“领域”对应）。

多范型设计并不期望得到一个经过深思熟虑的解决方案；这主要是实践性和作用域的问题。第 9 章所讲述的模式被限制在结构的共同性和差异性上，它们只是 C++结构的简单扩展。

1.9 小结

选择设计方法是一个关键的软件开发决定。在对象范型流行的今天，我们必须推广对象以外的范型，并寻找应用这些范型的机会。将自然界和业务中的每样东西都看作一个对象并不是最好的选择。我们可以使用共同性和差异性分析，同时在应用领域和方案领域中寻找共同性和差异性的维度。这些维度形成了抽象的族。

为了开始使用多范型设计，我们现在继续描述“分析”。下一章将讲述如何为应用领域构造一个起步“语言”基础：捕获了相关的族的一个领域词典。然后探讨如何让 C++更自然地表达族。目标是在应用领域族结构和适合的解决方案结构之间找到自然的匹配。随着本书的继续深入，我们的代码将更有希望满足预期的需要，并继续长期满足应用领域的需要。

第 2 章
共同性分析

共同性和差异性是大部分软件范型的两个关键特性。本章介绍共同性。共同性定义了一个应用中的“族”（family）。族是多范型开发中抽象的主要基础。

2.1 共同性：抽象的本质

软件设计中所有主要的进展都建立在从抽象中新获得的知识基础上。普通英文中，抽象的含义是只注重一般的东西，而抛开具体的东西。我们所说的抽象强调什么是共同的，而不注重细节。抽象是基本的分析工具。

假设我们正在设计一个处理工资单数据的程序包。我们知道要先根据很多不同的标准来对数据进行排序。在某些层次的设计中，可以将“排序法”（sorting）记为一个关键的业务抽象。它不是从对象范型中得到的抽象，而是从程序范型中得到的抽象。然而，它是一个关键的业务抽象，而不是一段简单的代码。如果知道数据基本上都是以随机顺序提供给排序算法的，那么我们很可能会使用快速排序。可以用 in-place 排序法对已经部分排序的大型数据集合进行排序。如果内存允许，可以将这些数据复制到一个链表中，然后使用链表插入排序，完成排序后再复制回来。这种方法通常是 in-place 排序法速度的两倍。如果我们只想为向学生练习用而编写一个名字有趣的函数，则可以使用“冒泡”排序。在高层次的设计中，因为这些排序方法存在共同性，所以可以同等地看待它们：它们都是“排序”。此共同性是抽象的基础。

如果要建立一个“矩阵代数”（matrix algebra）程序包，我们就要讨论抽象“矩阵”，尽管我们知道存在很多种不同类型的矩阵，如“稀疏矩阵”（sparse matrix）、“单位矩阵”

（identity matrix）和“上/下对角矩阵”（upper and lower diagonal matrix），它们的实现各不相同。但是根据它们表现出的共同的行为特性，我们将这些矩阵分为一类。可见，共同性又一次支持了抽象。

共同性分析与耦合和内聚这两个重要的设计原则紧密结合一起进行。软件设计竭力创建在很大程度上独立的设计产品，这些产品可以单独地进行建立、实现和维护。一个成功的划分会让设计产品在很大程度上去耦（以便开发团队更独立地工作）和内聚（每个设计产品都是一个概念整体，从而使系统更容易被理解）。例如，我们试图建立一个“继承层次结构”（inheritance hierarchy），它将在其他继承层次中独立的类组合在一起。这些层次结构从共同性分析中产生。尽管共同性分析在面向对象的方法中是隐式的，但在多范型设计中，我们把它变成显式的。因为这些类拥有共同的属性：共享共同的接口，并且可能共享共同的实现，所以将它们组合在一个继承层次结构中。其他的语言特性捕获其他种类的共同性。例如，我们可以使用模块将代码和数据结构大体相似而实现不同的结构组合在一起。

共同性分析为我们完成了三项重要的任务：第一，它提供了抽象。通过在单个设计抽象下将很多类组合在一起，它帮助设计者将设计分成各个“组块”（chunk），将问题化繁为简。继承层次就是一个很好的例子：基类代表了层次中的类的抽象。第二，共同性分析提供了内聚和耦合。依据共同性分组的耦合自然导致了组块的产生，因为每个组块与其他组块中的元素共同性很少，所以这些组块之间是相互独立的。第三，共同性分析缩减了维护的费用。共同性越宽泛，共同性分析所需要的时间也就越长。健壮的共同性可以成为设计中不变的结构。例如，如果按照三种模式定义基类 telephone：`alert`（用于振铃）、DN（用于获得电话号码，行业术语称为“directory number”）和 `talk`（用于将电话连接到一个通话通道），我们将获得这样一个抽象，它描述了现今存在的电话终端及过去 100 年间大部分电话终端的特性。我们可以将附加功能性放在派生类中，但是基类捕获了不随时间、空间变化的领域抽象。

2.1.1 演绎和归纳共同性

我们通过两种方式来认识共同性：认识我们以前见过的模式以及我们还没见过的递归结构。前者涉及到“经验”（experience），后者是一种“学习”（learning）的形式。两者都与软件设计有关。复用是所有关于经验的内容，而剩余的设计工作——创建新的抽象——则与学习模型有关。我们认识软件的方式与我们认识周围世界的其他事物的方式并无太大的区别。

日常生活中见到一辆汽车时，我们都能认出它是一辆汽车。即使看到的是至今尚未

见过的一款新车，我们还是认出这些“实体”就是汽车。这些汽车与我们经验中的汽车在细节上并不相同，但却拥有作为汽车的共同方面。看见这些车时，意识中的“原型车”帮助我们认出了它们。这些“原型”塑造了我们所建立的新系统，而不必根据最初的原则去发明“车”的一个新概念系统。

软件中也存在相同的道理：我们可以将原型或“框架”（frame）作为设计的基础[Winograd1987]。在遇到一个计算问题时，大部分人都会从经验中提取出抽象。很多人将这个问题抽象为带参数的过程，还有一些人将其抽象为数据库关系或元组；而大多数人会将其抽象为类和对象。所有这些都是“归纳”的抽象方法。很多方法只是从单个范型中抽取一些规则，将一个复杂的系统抽象为一个简单一些的模型（“如果你是一个锤子，那么每个问题看起来就像是一个钉子”）。

如果面对的是新情况，没有提供任何已知模型，我们就通过寻求“再现”（repetition）来形成抽象。由于缺乏类比和经验，我们应用“演绎”推理法来形成模型和抽象。在看见具有异国文化（与我们自己的文化不同）的房屋的第一时间，我们逐渐理解，这些房屋与文化之间有着某种程度的关系。从而形成我们将其标记为“那种文化的房屋”的抽象。同样，在软件中我们也可以这样做。

多范型设计利用可用领域的经验，将问题划分为可以被普遍接受的多个领域。随着设计的进步，多范型设计越来越依靠领域自身，而不是使用常规划分。对新领域的设计绕过了最初的直觉阶段，只在需要的时候借用原型，并注重从领域自身中找到划分的线索。我们从领域自身中提取出抽象，而不是使用预先形成的划分标准从问题中艰难地获得抽象。如果常规划分正好是给定应用的最佳选择，共同性和可变性分析将支持这个结论。

这并不意味着每个新领域都需要一个定制的范型以派生出其解决方案的结构。划分的形式有多种，我们称之为“共同性类别”（commonality category，第 2.3 节），它们一次又一次地浮现出来，设计者应当认识这些类别并利用它们。这些共同性类别中的大部分映射为 C++编程语言的结构以及这些结构所支持的范型，如对象和模板——这可能是这些结构存在于 C++编程语言中的原因。尽管如此，还是可以枚举出共同性的一些“公理化”（axiomatic）的维度，它们在很多不同的世界观中却是共同的。这里，我们选择的角度并不构成一个完整的体系，此选择也绝不惟一。这里选择的维度将形成我们对 C++实现方法（第 8 章）的选择。

2.1.2 软件族

软件问题以及这些问题的解决方案的结构都很丰富。可以将数据结构和功能按照标

准（诸如结构、名字或者行为）组合起来。包含关联项的某个组称为“族”（family）。Parnas[Parnas1976]对族的定义为：

一个程序集合构成一个“族”，从集合的“第一个”共同属性开始，“然后”确定族的个体成员的具体属性，这样研究程序总是很有价值。

我们可以在大部分的应用中发现很多族的抽象。

下面以线性代数编程的领域分析为例进行分析。该领域产生一个用户抽象的族，称为“n 维矩阵”，它包含列向量、二维矩阵及高阶矩阵。根据对这些程序的经验，我们需要一个数据结构族，以实现对这些数据结构的多样的内部表示：稀疏矩阵（可能使用链表）、索引顺序数组、单位矩阵及对角矩阵。还有一个族是矩阵乘法的算法集合：一个用于稀疏矩阵的乘法，一个用于单位矩阵与另外一个矩阵（一个经过处理的高效率的函数）相乘，一个用于有一个操作数为单位矩阵的情况等。其他运算也有类似的族。深入的分析还可能得到更多的族。

这个矩阵领域（可能构成某个更大问题的一部分）也有其自己的丰富的族。大的矩阵领域可以分成多个子领域：矩阵、数据结构和运算符。每个子领域都要进行自身的共同性分析。

很多的设计方法都集中在模块上。模块的种类有多种。对象、类、过程和函数都是常用设计范型产生的模块。模块是系统抽象、管理以及配置的主要单位。它们不需要对领域进行描绘。在某些业务概念的逻辑聚合与用编程语言捕获语言结构的能力之间存在一个张力。

大部分的传统软件设计是自上而下的模式，而对象范型既不是自上而下的，也不是自下而上的，它仍然紧紧依靠层次。我们根据部件进行分类，并将这些部件按层次组织起来。结构化分析、功能性分解、模块信息隐藏及面向对象的设计的共同性在于它们都产生“部件”（part）。使用这些部件可以隐藏设计的“秘密”。只要领域允许在模块中隐藏设计秘密，我们就应该这么做——面对变化时，设计将更具“弹性”（resilience）。但复杂系统往往有多个“顶端”（top），所以自上而下和层次设计范型有时无法描述某些领域。

矩阵领域就是一个例子。我们不是在传统模块中捕获重要领域的维度——也就是对应于源程序或对象代码的连续块的结构——而是试图在语言结构中捕获它们。这些语言结构可以跨模块应用。例如，可以在模块族中捕获矩阵结构，让稀疏矩阵、单位矩阵、上下对角矩阵等都有各自对应的模块（尽管可以使用类替代我们认为不像支持实例中模块的模块）。对矩阵的操作也是一个族、领域，但它们是重载族，而不是类或模块。

找到领域

从这一点出发，有洞悉力的读者将认识到，我们可以捕获带重载的算法的共同性、

带共有继承的行为的共同性及带聚合的结构的共同性。但首先需要找到领域，这是共同性分析的第一步。在这方面有经验的人可从记忆和经验中获得抽象，这样大部分的领域对于他们都很明显。在新的问题范围中，需要收集关于问题的信息，从而将问题划分为多个领域和子领域，并描述它们的属性。为了方便大家入手，我们开发了一个术语词典。使用这个词典，我们解释了领域的每个“片断”的意义，以及整个领域的意义。

从领域的定义开始，帮助我们将一个问题分成可单独管理、可实现和可复用多个方面。再以矩阵领域来分析，设计者必须创建结构来存储和组织矩阵中的数据。为了“交换”（swap）“调页”（page）大型矩阵到磁盘上，并交换或调页来自磁盘的大型矩阵，我们必须知道如何在主存储器和辅助存储器之间来回映射。一个“顶端”表示领域；另一个“顶端”表示矩阵自身。这样将矩阵分成两个子领域——一个用于表示矩阵，另一个表示存储器管理——从而形成了这样的一个架构，在此架构中，若将有关问题一起考虑，其中的模块将耦合得更紧密，并具有更加独立的可管理性。

下一节讨论设计者如何使用词汇表尽早区分出设计中的各个领域。

2.2 起动分析：领域词汇表

词汇表提供了一个领域模型的结构并说明所需的基本“构建块”（building block）。单人程序组可以轻松地将一个应用词汇表收集起来供自己使用，不清楚的词汇可以暂缓，等到程序员选择处理它们时再行解决。但多数引人注意的软件任务——包括多数作为单独实现成果的任务——都需要多个人之间（程序员、客户、测试人员、市场人员、软件设计师、硬件人员等）的交流。这些人必须相互交流，以制定所出现问题的解决方案。此时，对工程交流效率的提高来说，存在一个共同的词汇表是很必要的。即使是单个的程序员，也可以避免设计的群体性，但在努力实现从分析概念到解决结构的转换时，又将涉及到词汇表的问题。

2.2.1 领域词典

我们通过建立一个“领域词典”（domain dictionary）记录应用领域的词汇开始了共同性分析，目的是获得正在解决的问题的所有相关方面，以及解决方案的可能方面。在细致、深入地理解问题之前，我们逐字和逐个短语地对问题扫视一遍，这些字和短语来自客户、要求文档及我们的经验，我们称之为“领域词汇表”或“领域词典”。它是很早就开始使用的软件设计工具，并在当代的设计方法中获得了各种形式的复兴（相关主题的详细信息请参见[GoldbergRubin 1995]）。

领域词典是我们所工作的问题范围或领域中的技术术语的目录。它与用于支持结构化设计方法的数据词典类似。当最初遇到问题时，我们就面临着即时的需求。多数的软件开发都由客户的明确的问题来驱动。这些应用的特性是具体的。大部分问题语句的“具体”（specific）特性妨碍了我们发现它们的共同性。客户群很少一起考虑他们所面临的共同问题，所以他们不大可能向软件开发者提出“通用”（general）的需求，以获得“普遍”的解决方案。但作为分析者和实现者，我们需要研究该“业务”是否适用于具有类似抽象的其他应用。我们需要开阔视野，以增加发现跨领域的共同性的可能。我们对应用族进行设计，将手边的最初应用作为族的一个原型成员。对一个明确定义的领域检查得越充分，发现的共同性就会越多。要注重“领域分析”，而不仅仅是一般的分析（普通英文意义上的分析，或典型软件设计方式中使用的分析的意义）。

领域分析相对于简单分析来说有两个主要的好处，尽管这些好处是相互对偶的。第一个是“通用性”（generality），它可以支持复用。如果要为某个信息协议应用建立一个FSM（finite-state machine，有限状态机），我们可以实现一个满足特定需求而又不排斥其他需求的通用FSM。这种通用性是有代价的，需要花费更多的精力去研究FSM通常的属性，而不仅仅是了解单个的应用。但从长远来看，如果相同的抽象能被复用，那么这种代价还是值得的。

领域分析的第二个好处是其面对变化时的弹性。如果已经实现了单个客户原始的设计说明，但为适应客户实际情况的变化，又需要（对解决方案）加以修改。但如果对设计进行拓展以满足类似客户的需求（即使这是一种假设），我们考虑一下将会发生什么情况。随着原客户需求的变化，实际上等于产生一个新客户，新客户的需求与原设计说明中的需求类似。如果我们设计得更加宽泛，就可以比优化手边的应用更从容地适应需求的变化。

设计者必须理解：不仅共同性是跨时间、空间稳定的，变化趋势也是跨时间、空间稳定的。我们可以使用基于差异参数的模型来描述变化的特性。差异参数值的变化范围产生了软件族的成员。第3章将讨论“差异性分析”，它是建立这种差异的总体框架的方法。如果软件的差异性分析完善了共同性分析变化的通用性和弹性，那么领域分析将带来更加灵活的软件。如果使用领域分析设计一个FSM，从而使抽象足够健壮（通用），并且变量参数可以有效地预期变化的范围，那么这个FSM就可以对跨越很大范围的不同应用（复用）进行定制（以满足不同用户的需求），而无需在运行时支付由通用性所带来的花费。在实践中，共同性和差异性分析实际上是无法分开进行的。灵活性、对变化的弹性及通用性也是密切相关的。

下面考虑建立用来分析FSM的词汇表。我们可以分析需要建立的第一个FSM——也

许是能够支持某个具体消息协议的 FSM——对应的词汇表。但多个 FSM 形成了一个族。同样，服务于一个消息协议的基本 FSM 结构，也应当是 GUI 中的对话框序列的基础。我们需要捕获 FSM 层次上“共同”的地方。

所有的 FSM 究竟有什么共同的地方？他们都有一个当前“状态”（state）。该状态被实现为“自定义类型”（user-defined type，比如枚举或整型）的一个实例。他们都有概念上的“转换表”（transition table），它是从“当前状态”（current state）和“输入事件”（input event，这些事件中的一个族，同一个“事件类型”（event type）中的所有事件）到新的当前状态和“转换函数”（transition function）的一个映射。

当代面向对象的分析也从领域词汇表开始，活动的输出通常也称为“发现对象”。对于第一个近似值来说，此活动的输出是类的列表（所以“发现对象”是一个无伤大雅的误称，但可以参见[GoldbergRubin1995]以了解一种不同的策略）。来自面向对象的分析的领域词汇表不仅包含类名（比如 `Stack` 和 `Window`），还包括给予 `Stack` 和 `Window` 意义的其他名称：`push`、`pop` 和 `top`，`displayText`、`refresh` 和 `clear`。多范型设计中领域词汇表的运用是对象的“发现对象”运用的一个扩展。

2.2.2 领域词典团队

开发领域词典是一门艺术，它为领域定义了设计的语言。开发领域词典应当是一个团队活动。CRC card[Beck1993]提供了一个如何用多个设计者的观点来建立一个领域词典的例子。当 CRC card 的抽象聚焦在对象范型上时（比“我们必须进行多范型分析”更缺乏创见的观点）它们为社会化设计提供了一个好的模型。

领域词典的开发是一个迭代活动。当词典的规模逐渐变大时，让领域词典的开发团队暂停下来对已获得的进展进行内省是很值得的。我们可以问问自己关于先前完成的领域词典的一些重要问题。

- 定义是不是清楚？架构团队、客户和用户对这些术语的意义是否有共同的认识？
- 这些定义都用到了吗？所有这些术语都适合预期的应用领域吗？设计团队有这样一种倾向：脱离目标领域、超出目标业务需求，并进入最好不要涉及的区域。词汇表可以帮助我们识别这些区域。
- 所有需要的条目都进行定义了吗？
- 这些定义与需求文档中出现的术语相符合吗？如果没有用户合同（而不是客户合同），并脱离了需求文档的语境，这些领域词汇可以有自己本身的意义。与用户一起经常性地重新审视领域词典，可以确保需求文档语境中的这些术语有着前后一致的意义。

这些问题有助于激发思考，而不会约束过程。例如，领域词典团队可以不尽心进行精确的定义。相反，可以通过例子和比喻来描述领域的特性。初步的特征描述甚至可以忽略用途和基本原理的说明。例如，一个最初的 FSM 领域词典可以像下面这样：

- AbstractFSM：考虑跨连接实现多个消息协议的一个应用。一个 FSM 将实现各种协议。因为普通的应用逻辑可以通用地看待 FSM，所以全部的 FSM 都可以交换使用。AbstractFSM 领域在这个层次上捕获共同性。
- ImplementationFSM：所有的 FSM 都将有一个实现，以使用表或其他数据结构对形式（当前状态、激励、转换动作、下一个状态）的元组进行表示和管理。此领域捕获了这种共同性。
- UserFSM：每个 FSM 都有对具体转换动作的定义，以及用户提供的其他代码。此领域提供了定义和管理这些元组的场所。
- State：状态可以表示为枚举、整数或一个抽象地表示更复杂的应用状态的类。
- Stimulus：激励是 FSM 模型基础的四元组的元素之一。对于每一个状态，激励使机器按序列进入下一个状态，同时以副作用的形式执行转换函数。它潜在的同义词或替代名称是 **Message** 和 **Event**。
- TransitionAction：映射当前状态和激励下一个状态的函数族。每个函数体实现状态转换的副作用。

第 8 章中我们详细阐述设计时将使用这个领域词典。

在 FAST 中，Weiss[Weiss1999]通过使用斜体字表示领域词典术语以注释需求文档。

共同性分析是面向对象分析的泛化。我们不只对有意义的类（比如时间类型和状态类型）感兴趣，对共同的结构（转换表和当前状态）、用例（外部 FSM 接口和支配用户的布景和规则）以及共同的名字和行为（状态转换函数，它们都共享共同的行为、参数和返回类型）也感兴趣。对领域词典来说，所有这些抽象概念都应该受到平等的对待。在后面的章节中，我们将更详细地探讨软件共同性的这些方面。

2.3 共同性维度和共同性类别

设计是一种创新活动。这种创新主要源自我们如何使用刻刀将抽象雕刻到系统之中。我们可以首先在应用领域进行这项工作，然后是方案领域。但实际上大部分人通常会在两个领域同时进行。如果我们在应用分析期间对方案领域加以预见，那么我们可以更轻易地将问题的结构表达为一个 C++解决方案。

设计远不止将一个扁平的面饼切成饼块那么简单。程序的模块化结构只是在设计和

实现中我们可以捕获的一个维度。在进行设计划分时，我们可能需要同时依据多重标准来进行划分。

让我们考虑如何在设计中分离出类。我们可以取一组显然无关的类（因为它们的成员函数都不同），然后将它们按照某种线性序列编排。我们可以根据重要性，或者根据名称的字母顺序，亦或按照我们想要的顺序，先将它们在同类别（akin）的层次进行分类，Booch 称之为“类类别”（class category）。除非我们试图减少划分形成的集合之间的共同性，否则如何区别类类别的详细内容并不重要。例如，在一个线性代数包中，我们可以有诸如标量数字和矩阵这样的简单类类别（如图 2.1 所示）。

图 2.1 标量数字和矩阵类类别

现在来考虑向这个分类方案中添加派生类。因为某些类的成员函数的签名非常相似，所以我们想将它们一起放入到一个继承层次中去。此时不能再将它们放在同一条线上；这种分类存在于另一个“维度”中。[①]参见图 2.2。

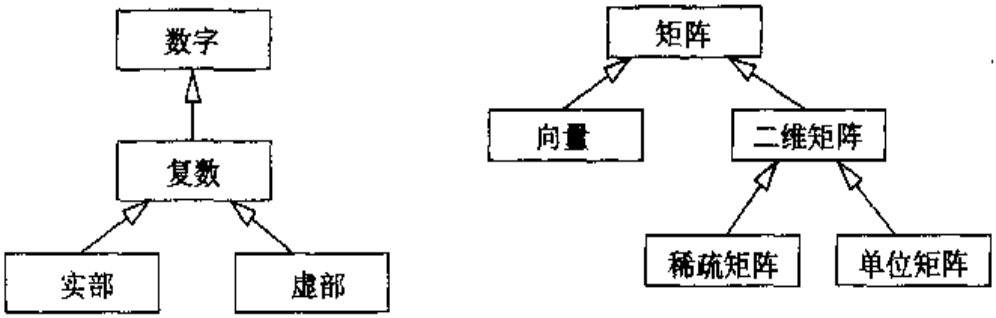


图 2.2 捕获签名共同性

区分基类的维度垂直于分离基类和派生类的维度。

也存在一些自由的函数，它们带有多个参数，不同的参数依据不同的类类别进行声明。因为每一个这样的函数都可以存在于多个不同的实现中，所以它们中的某些或全部都可以重载。这就意味着函数形成了族。这些函数还对类产生影响。它们与用于区分类类别和安排继承层次的成员函数具有相同的设计“高度”（stature）。一个很好例子就是某些代数层次上的 operator*函数族（Vectors 是列向量），如图 2.3 所示。

① 比方说，如果开始类类别“数字”和“矩阵”是在一个横向的维度上，那么此时添加派生类就可以在竖向的维度上进行。——译者注

```
Vector operator*(const Matrix &, const Vector &);
Matrix operator*(const Vector &, const Matrix &);
Vector operator*(const Vector &, const Vector &);
Matrix operator*(const Number &, const Matrix &);
Vector operator*(const Number &, const Vector &);
Number operator*(const Number &, const Number &);
. . . .
```

图 2.3 operator* 函数族

此函数结构不能完全同类结构分离开来。但是，它也没有与类结构对齐。我们可以将其看作设计结构的第三个维度。

到此为止，我们还没有论及类的内部结构，它可以形成一个不同于以签名来分组的分组。如果最初的抽象是模板而不是形式相近的类，那么每个模板参数就有一个附加维度，如图 2.4 所示。

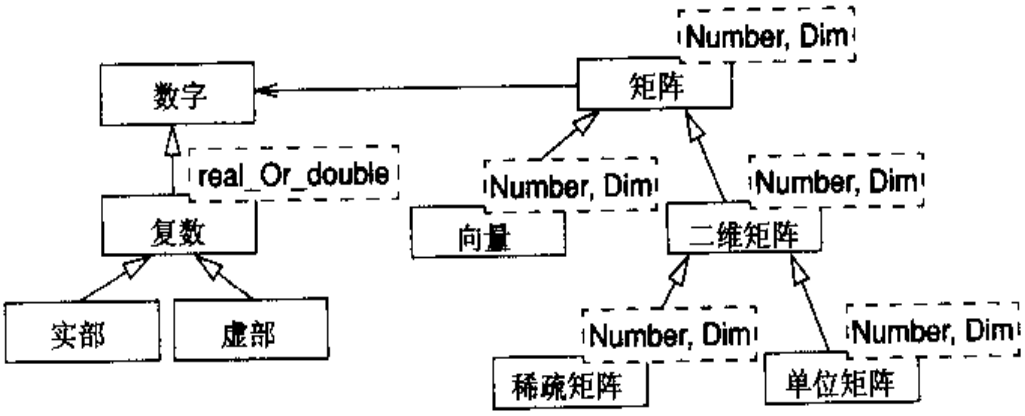


图 2.4 另一个维度：模板参数

如此不停地进行下去，结果就很复杂（或者说很丰富），建立了设计空间模型的多维空间呈现在 C++程序员的面前——这比类或对象的简单二维平面要复杂得多。实际上，设计空间（包括问题领域和方案领域）是相当复杂的，因为这些维度并不都是相互垂直的。例如，重载成员函数与继承之间通过复杂的方式相互影响。

设计是一门艺术，就是要从这个空间中“雕刻”出有用的、富于表现的和可维护的结构。我们所使用的“刻刀”必须能够刻画出一个清晰的划分——同时多范型设计为管理这种设计空间的复杂性提供了一套方法。针对此多维设计空间的各个方面，我们将使用术语“共同性的维度”（dimension of commonality）。共同性的范例维度包括：

- 结构（Structure）
- 名字（Name）和行为（Behavior）
- 算法（Algorithm）

这些维度描述了各种共同性，我们正是使用这些共同性将族成员分组到一个子领域中的。但只进行共同性分析，还不能选定一个适合于问题结构的解决方案的结构。我们可以使用这些维度来描述各个族成员之间的“差别”。例如，所有矩阵的行为都相同，但它们拥有不同的内部数据结构。对族成员的差别所进行的研究称为“差异性分析”（variability analysis），这是下一章的主题。

这两个维度（一个针对主要的共同性，另一个针对主要的差异性）在一起描述了一个“共同性类别”（commonality category）。与对“绑定时间”（binding time）、“例化”（instantiation）和默认值的要求结合在一起的共同性类别指向一个惟一的C++语言特性。这些语言特性都是设计用来表达特殊范型中的抽象的。共同性和差异性分析帮助设计者形成了软件族，还帮助设计者为每个族选定了设计范型。

假设矩阵的例子要用于运行时间绑定，共同的行为和不同的数据结构指向带虚函数的继承。这就暗示了一个特殊的范型——这里是指对象范型。每个C++语言特性——模板、#ifdef、继承或带有虚函数的继承——都对应于共同性类别、绑定时间和控制例化的可选参数的结合。

在实践中，共同性和差异性分析很少分开来使用。实际上，差异性经常源自共同性。所以当我们为了获得沿共同性维度的相似性而研究应用领域时，我们可以同时沿着相同的维度来寻找差异性。差异性的维度与我们对共同性所使用的维度相同。例如，我们注意到：不同种类的矩阵在结构上存在不同，而继承层次中的类则在它们的有些成员函数的名字上及名字相同的函数的算法上存在不同。

总的来说，共同性分析是非正式的。本书中介绍的方法使用了简单的表格来组织抽象和记录分析。这些方法没有使用算术上正式的因子分解，也没有使用CASE工具。相反，我们依靠熟悉目标领域的优秀设计者的直觉和见识来解析出设计中共同性的惟度。我们先非正式地写下它们，在工作进行的过程中记笔记。最后将共同性和差异性分析的结果绘制成表格，以支持后来的分析。

下面三节描述了我们用来定义共同性类别的共同性的基本维度：数据、名字和行为，以及算法。

2.3.1 （数据）结构

在多数好的设计中，数据结构比操纵它们的算法演进得更慢。这可能是因为数据更接近地反映了问题的结构，并且它本身比过程更稳定。或者是因为数据变化的代价迫使程序员避免直接改变数据。在这两种情况下，好的数据设计对可维护的系统来说都很关键。为保持随时间的稳定而追求适当的数据抽象的倾向，是对象范型宣称的胜过基于过

程设计方法的优点之一。数据结构是共同性的一个重要维度，尽管它不是惟一的维度，也不应当与共同性的维度中面向对象的设计的主要维度相混淆。

数据设计是经典设计方法（比如结构化分析及其派生物[Yourdon1979]）主要关注的部分之一。在交互式用户界面以及随之而来的计算的事件驱动方法出现以前，数据结构和控制流是设计的主要焦点。批处理工资单系统中最重要的抽象是数据结构和不考虑这些数据的过程序列。在当今流行的很多设计方法（如 Shlaer-Mellor，它们很早就开始考虑数据结构和数据流）中，依然可以找到聚焦在数据上的经典设计方法的影子。Rumbaugh 中的记法和方法都表明：尽管数据不是主要的焦点，但我们应先理清设计中的结构关系。

交互式计算近期的趋势已经将软件设计的焦点从过程和数据“流”（flow）转移到数据“结构”（structure）和行为上。批处理应用或保持其原有的地位或逐渐衰落，而很多新的软件系统却支持交互式界面和“反应性”计算。这些系统必须从一个实时的数据链接或者从穿过 GUI 的光标移动中对随机序列的数据查询产生反应。在时序系统中，尽管数据只是“凑凑热闹”，但数据流却暗示了任务的顺序。在交互式系统或实时系统中，系统必须发送服务请求给能够处理它们的代码。计算的模型不再是通过有序过程的平滑数据流。相反，它发生在以无法预期的顺序零星猝发的活动中。在对象范型中，我们通过将相关的活动猝发按照语义分组（称为“成员函数”）来创建抽象。这些抽象（称为“类”）的核心部分通常有一个数据结构。但是，我们更感兴趣的是这些数据是如何将有关的操作联系在一起的（正是这些操作形成了抽象），而不是数据本身或这些数据是如何“流过”系统的。我们称这种为抽象一个“抽象数据类型”（abstract data type）。

当前基于角色设计[Reenskaug1996]的趋势进一步强调了抽象的“本质”与给定环境中（它的角色）其功能性行为之间的区别。在多数职责驱动、面向对象的方法中，行为共同性驱动了多数重要的抽象。我们努力将结构推迟到设计快要结束和实现开始的时候。CRC card[Beck1993]和 Wirfs-Brock 的职责驱动的设计[Wirfs-Brock1990]就是这样的例子。在这些方法中，我们聚焦在形成“系统”的总体分组上，而不是聚焦在形成“代码”的数据和算法的细节上。尽管有时数据也驱动了设计结构，但上面的分析仍暗示我们应放开眼光，将数据结构看作划分标准的主要来源。

多数 C++程序员仍然将类和继承层次与它们的数据设计紧密联系在一起。常常将对象看作“智能数据”，而不是有关行为的轨迹。对象范型的支持者们很早就冒然断定：对象维护的好处根源于这样一个“老”原则：数据比过程对时间更加稳定，类只是表达这种原则的一种合适的机制。在现代交互式用户界面和实时系统充斥的年代，继续强调这种观点的多数方法有它们自身的“根基”。它们是“新经典”的方法，它们表面上使用一些对象原则，但在数据和过程分析中它们拥有自己的“根基”。但老的方法很难消亡。今

天使用的一些流行的设计方法自觉地推迟了对数据布局的考虑。现代的方法有一个共同的“故障模式”：它们都将成员数据映射成为一个状态集，并将所有的对象当作 FSM 来对待。FSM 可以是一个对象（有关详细内容，请参见第 8 章），但很多类并没有被很好地建模成状态机。例如，Window（窗口）是一个很好的类（有关响应的轨迹）但它几乎不是一个状态机，我们对其数据并不太感兴趣。不熟练的设计者除了专注于当代设计方法中的数据以外，还常常聚焦在数据存储（比如报告或表的数据），以便发现对象，并用“给予数据智能”的成员函数来标记它们，将数据装配到对象范型中。

有人认为数据稳定性与对象稳定性之间存在对应关系，但这经不起仔细地推敲。我们来考虑类 Number 及其派生类的情况（参见图 6.2）。行为层次是清晰的，紧接着是直接从应用领域派生出来的子类型关系。类 Number、Complex 和 Real 的意义，以及它们之间的关系都很清晰。简单的分析就可以得出一个类层次，该层次中 Real 是从 Complex 中派生出来的。不知情的设计者将会把 Complex 类中的复数封装为一种表示法，这就导致 Real 继承了很多它不能使用的多余的东西。

这里我们可以吸取几个教训。一个主要从数据结构派生出来的设计可以与基于职责分布的设计差别很大。数据结构仍然是一个重要的设计事项。尽管数据结构是实现结构，但好的设计将先行了解实现是如何形成设计决定的。要摆脱过于期望统一数据分析和对象分析的“新经典”方法的陷阱，我们需要将共同性的这两个维度分离开来，并在设计期间分别处理。

在正式开始之前，我想即刻指出：数据设计通常与一个好的对象划分是相对应的。这里想要指出的是我们不应当“滥用”这种调整。相反，我们应当诚实地获得它。

有一种完整的范型适合研究数据结构及其访问，那就是数据库。在很多数据密集型应用中，关系建模和其他基于数据的建模技术占有一席之地。（面向对象的数据库实际上并没有归属于同一类，因为就自身而言它们不是一种设计体系，而是在存储介质中持久使用的面向对象的设计结构的一种方法。）C++并没有直接表达数据库关系，所以本书在处理多范型时将它们看作“外部范型”。之所以称之为“外部”是因为它们没有进入编程语言的范围。但是，如果“在外部将它们与编程语言联系起来”，那么编程语言可以接纳它们。例如，我们可以将 C++程序与数据库管理库连接起来。C++不支持某些范型，这并没有减少这些范型的价值和 C++的价值。两者都将放在第 7 章的多范型方法的语境中进行考虑。除数据库以外，我们还将聚焦支持简单数据共同性的 C++机制，尤其当它在早期的设计中作为第二位的关注事项出现的时候。

我们来考虑数据结构共同性的一些例子：

- 磁盘文件系统。一个操作系统可以支持多种文件：数据库、无格式字符文件、格式

化记录文件、编入索引的序列文件、成块文件、非成块文件及其他文件。所有的这些文件都必须工作在一个主要“目录和文件结构”设计的语境中。多数文件共享基本的数据属性：创建和编辑时间字段、映射逻辑文件结构到基本磁盘存储块上的结构等。需要注意的是，数据结构可以（也可以不必）跨越文件系统行为的主要分组。

- 协议设计。消息格式是通过长度（如同在固定长度的包中转换和传递）、细微结构（位流或8位字节）、头格式（字段、长度和偏移量集）以及其他参数实现规范化定制的。数据共同性通过可分离的两种分析实现了消息的语义。
- 当我们构造软件来审查一个容错控制系统的数据结构时，我们注意到，多数可审查的数据都与父、子和同属结构相连接。尽管并不是所有的数据结构都拥有全部这些字段，但早期的分析仍然引起我们对它们的注意，所以我们将它们也包括在领域词典中。在共同性分析期间，我们可以更精确地描述围绕此共同性所形成的领域的特性。我们也必须使用第3章描述的差异性分析的方法，来处理数据结构之间的不同。
- 有些内存管理算法根据长度将数据分组成等价的类，即将所有同样长度的存储块放在一起管理。
- 数据分析应用将所有的标量放在一起处理，所有的有序偶（复数、直角坐标上的点、极坐标上的点）放在一起处理，所有的有序三元组（直角和极坐标的三维点）放在一起处理等。

在主要关注数据结构的同时，我们还可以考虑源代码的结构。模板函数有一个可以生成多种不同实现的单一、共同的结构。例如，下面的函数捕获了一个排序算法的结构，它是一个跨多数可排序类型的共同结构[①]：

```
template <class T>
void sort(T s[], int nelements) {
    // a good sorting algorithm written in terms of T
    for (int top = 1; top <= nelements - 1; top++) {
        // vector is sorted in positions
        //       0 to top - 1

        register int j = top - 1;
        while (j >= 0 && s[j+1] < s[j]) {
            T temp = s[j];
            s[j] = s[j+1];
            s[j+1] = temp;
            j--;
```

① 也就是对多数类型数据的排序都适用。——译者注

```
        }
    }
}
```

模板函数能够很好地处理少量的共同性和差异性。例如，它们可以处理这样一种情况，即多数的算法都具有共同性，只存在可以通过模板参数进行模块化处理的局部差异性。从这种意义上来说，它们可以像 Gamma 所著的设计模式一书[Gamma1995]中的 Template Method 模式一样来使用。模板参数在算法模板结构中管理差异性，就像它们在类模板中管理数据结构的差异性一样。模板参数可以具体化为简单的数值，或者是复杂的自定义类型。

我们将在第 6 章中了解到：继承是表达共同的数据结构的最通用的工具，并且，模板同时可以捕获共同的数据和代码结构。

2.3.2 名字和行为

经常听说计算机科学中的很多有趣的问题最后都变成了“名字里是什么”，并可以通过其他间接的层次获得解决。名字传达了意义。我们可以使用名字的共同性将具有相同意义的条目分为一组。但一定要注意避免同音异义词的分析的类似情况。有很多种编程语言的名字：函数、类似、参数、数据标识符及其他。现代计算机科学都涉及到这些术语，它们通常拥有正式而“高贵”的根基。

除了所有的这些，我们还可以根据名字所提示的线索来进行重要的设计。随着设计和实现不断地向前推进，我们可能会发现需要用到下面这些函数：

```
void set_date(const char *const);
void set_date(Year, Month, Day);
void set_date(time_t secondsSinceStartOfEpoch);
```

因为这些函数都有相同的名字，在直觉的层次上有共同的意义，所以它们很可能形成一个族。这个例子显示了如何利用带有函数重载的 C++中这种共同性的优点。继承层次中的超越函数是另一种类型的函数族。

有一点很重要，那就是分析期间要先识别出共同性，并将语言特性的选择推迟到完全了解了共同性类别时再进行。例如，我们来考虑一个通用图形系统的设计，在此系统中，我们将发现一个名为 displayOn 的函数族。这些函数将对象显示在一些输出介质上。我们可以将 displayOn 实现为一个重载函数族，正如前一个例子中所做的那样：

```
void displayOn(BitMap*, XWindow*, Point);
void displayOn(const char *const, XWindow*, Point);
void displayOn(BitMap*, MacWindow*, Point);
void displayOn(const char *const, MacWindow*, Point);
```

或者可以将 displayOn 实现为每个类（类的对象是可以显示的）的一个成员函数：

```
class XWindow {
public:
    void displayOn(BitMap*, Point);
    void displayOn(const char *const, Point);
    . . . .
};

class MacWindow {
public:
    void displayOn(BitMap*, Point);
    void displayOn(const char *const, Point);
    . . . .
};
```

或者可以将它们实现为通过继承而相关的类族的一部分：

```
class Window {
public:
    virtual void displayOn(BitMap*, Point);
    virtual void displayOn(const char *const, Point);
    . . . .
};
class XWindow: public Window {
public:
    void displayOn(BitMap*, Point);
    void displayOn(const char *const, Point);
    . . . .
};

class MacWindow: public Window {
public:
    void displayOn(BitMap*, Point);
    void displayOn(const char *const, Point);
    . . . .
};
```

或许还有更多基于模板、或基于习惯语法或模式（比如“双重调度”，double dispatch）的选择。该选择主要依赖于绑定时间，但是也可由设计者的洞悉力和经验支配。

我们在需求文档中、用户词典中，以及预先存在的代码中发现名字。为了进行分析，所有的这些来源都是我们涉猎的对象。通过在这些来源中搜寻名字、结构、序列以及它们之间的趋势和关系（这更加重要），我们就开始了共同性分析。这些趋势和关系形成了族，并且我们可以为抽象中的这些族捕获“遗传密码”。

通过名字分组的族的类型有多种。函数集合和模式都可以有名字。有一点应当清楚，我们要对它们进行单独的分组。我们可以定义一个新的术语（如下所示）对共同的“名字族”进行分类，这些术语凭借自身的权利成为共同性的重要维度。

- 标识符：标识符是某些语境（context，比如作用域）中惟一的名字。按照惯例，它是一个数据或一组数据的名字。更一般来讲，函数的名字或类型也是标识符，尽管后面的章节中我们是分开来处理它们的。
- 签名：签名是操作、过程或根据参数和返回类型来描述的函数的接口。通常我们将一个名字与一个签名关联起来。函数声明文档称为签名。签名包含行为和语义。例如：

```
Complex &operator+=(const Complex&)
```

这就是一个签名。函数族可以在高层次上共享同一个意义，但要用不同的算法来实现。签名传递了独立于任何特殊算法的高级的语义。实际上，一个给定的程序可以有多个与签名相匹配的函数。例如，每个函数都在其自身的作用域内匹配这个签名。

每个函数都有一个签名。拥有某个给定名字的所有函数并不一定要有同一个签名。比较明显的一种情况就是重载函数，它们是通过签名面相互区别的。具有相同签名和名字的函数可以形成一个族。很显然的一个例子是类层次中的成员函数。

大部分编程语言都从技术上将返回类型定义为签名的一部分。C++的技术定义并没有将返回类型作为签名的一部分，但我们可以推广签名的设计概念，从而使它包含返回类型。

- 类型：类型是一个签名集。它并不描述实现，而是描述行为。在某种意义上，我们应当将“类型”看作“抽象数据类型”（一个远离任何实现的接口的描述）。通常我们给予类型描述性的名字，比如 Stack、Window、TapeDrive 或 ComplexNumber。

在分析期间，有时我们可以直接发现某些类型，尤其是那些“明显”的类型（比如 Stack、List 和 Window）。更正式地来讲，类型描述了根据行为分组的抽象的集合或族的特性。在分析期间，我们可能发现根据行为分组的抽象，并将它们当成一个族，对签名集或类型中的这些属性编码。我们不应聚焦在族成员的数据结构上——那是一个不同的共同性维度。

注意：类属于在这里考虑的抽象之列。类是常用于实现一个类型的一种编程语言结构。但每个类将单个抽象中的共同性的两个维度（行为和结构）结合在一起。在分析的

初始阶段最好让这两个维度保持分离。如果将两个分析结合在一起进行，那么可能带来一个自然而单一的类结构。我们并不愿意过早地勉强得出一个类结构，从而对实现产生负面影响。当代的一些设计方法（比如 Fusion[Fusion1993]）清楚地认识到了这一点。

名字是面向对象的分析的一个重要组成部分。在多数流行的面向对象的方法中，我们主要聚焦在类名上。CRC card 帮助设计者聚焦在行为和传达行为的名字上，而不是聚焦在实现行为的数据结构和算法上。焦点是签名。

类型是定义了获得输入（类型签名中函数的参数）、产生输出（通过参数和返回值）和修改内部状态这些操作的抽象。类是类型的实现。类的实例（称为“对象”）连接在一个网络中，相互传递输入和输出，以获得整个系统的功能。

区分“行为”（behavior）和“意义”（meaning）是很重要的。这是一个细微的区分，而且很不幸这还是一个非正式的区分，但它足以帮助我们区分二者。行为与实现紧密地联系在一起。我们并不确切地关心一个对象的具体行为。例如，我们不关心实现类型行为的成员函数中确切的指令序列。类 `AsciiGraphicsWindow` 和 `XWindow` 的“刷新”（refresh）行为是不同的。但对客户端来说，两种情况下的刷新都“意味着”同一件事。客户端关心的是“意义”。程序可轮流使用具有相同意义但显示出不同行为的类。（第 5.1 节中讨论范型和对象时，我们会继续细致地讨论这个问题。）

2.3.3 算法

过程是最古老、应用也最广泛的编程的抽象之一。在历史上，过程的设计本质上具有很强的等级层次性，可以自上而下或者自下而上地进行。自上面下的设计方法聚焦在过程的分解上，它很少费力地将共同的算法或代码段合并成过程的层次的“分支”和“叶片”。共同的代码段经常被复制（针对链表或集合的代码，或者针对步入项目的数据结构的代码）在多人开发的项目中尤其如此。自下而上的设计解决这个问题时，使高层系统架构的观念变得不稳固，并被推迟。

如果对一个自上而下的设计进行共同性分析，我们可以发现形成族的代码段——用例。当我们通盘考虑整个系统时，这种共同性就会凸现出来，这与自上面下方案的设计者（他或她在某一时刻只能看到一个层次）的视角是不同的。在分析期间，我们要考虑执行线程和潜在线程的共同性。与在某一时刻只能正式地看到某一层的过程设计者不同，我们要找出高层的用例，在这种用例中，我们将整个系统做为一个整体来分析。与聚焦在作为执行单位的各个成员函数上的面向对象的设计者也不同，我们要考虑更长的执行链。

2.4 共同性的例子

本节介绍了针对结构、名字和行为，以及针对算法的共同性的例子。这些例子帮助说明了我们要在共同性中寻求什么内容。同时它们也表明：C++可以用不同方式来表达一个给定的共同性的类别。这种表达方式的不同促使我们在第3章中对差异性进行讨论。

2.4.1 结构

对大多数软件设计者来讲，结构的共同性依赖于直觉。借用一个简单的例子来说明：所有的 Shape（形状）都有一个中心点——类 Point。习惯上，C 语言是这样表达这种共同性的：

```
typedef struct _Point { int xpos; int ypos; } Point;

typedef struct _Circle {
    Point center;
    . . . .
} Circle;

typedef struct _Rectangle {
    Point center;
    . . . .
} Rectangle;
```

相反，C++程序员更倾向于使用继承来解析出这种共同性：

```
class Shape {
public:
    . . . .
protected:
    Point center;
};

class Circle: public Shape {
public:
    . . . .
};

class Rectangle: public Shape {
public:
    . . . .
};
```

也就是说，C++可以比C更直接地表达这种共同性。

C++给予了我们其他一些方法来表达结构的相似性。考虑一个List（链表）抽象的数据结构。所有链表在结构上都相似，如图2.5所示。

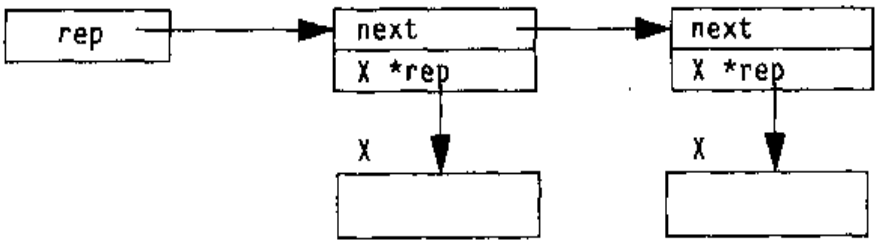


图2.5 链表的数据结构

但并非所有的链表都“相同”（identical），多处细节依据链表所拥有的各种元素的不同而不同（例如，每个数据块的实际长度依赖于确切的X值）。我们可以在一个模板中捕获这种相似性：

```
template <class X> class ListRep {
friend class List;
    ListRep<X> *next;
    X *rep;
    . . . .
};

template <class X> class List {
public:
    . . . .
private:
    ListRep<X> *theRep;
};
```

2.4.2 名字和行为

我们来考虑简单的领域 **Number**，我们发现该领域中有多个算术运算符的族，比如operator+、operator- 和operator*。每个运算符族都有“意味着”（mean）同一件事情的多个成员。例如，加法总是意味着同一件事情，而与其操作数无关。这种意义上的类似性反映在运算符的外部可见行为上，并在我们给予它们的名字中传达出来。我们使用“名字”（name）operator+ 来表示加法意义的这种共同性。

C++提供多种语言特性用以捕获这种共同性，不同的需要不同的C++机制。一个这样的机制就是一个重载函数。这里我们为在编译时绑定的分类操作数而重载operator+：

```
Complex operator+(const Complex &n1, const Complex &n2) {
    // would be declared a friend function in class Complex
    Complex retval = n1;
    retval.rpart += n2.rpart;
    retval.ipart += n2.ipart;
    return retval;
}

BigInteger operator+(const BigInteger &n1,
        const BigInteger &n2) {
    . . . .
}

Complex operator+(const Complex &n1,
    const BigInteger &n2) {
    . . . .
}
```

名字 operator+= 也表达出加法的意义，只是多了一个赋值运算符。我们可以使用继承层次中的 overridden 虚函数来绑定 operator+：

```
class Complex {
public:
    virtual Complex& operator+=(const Complex&);
    . . . .
};

class BigInteger: public Complex {
public:
    BigInteger& operator+=(const Complex&);
    . . . .
};
```

注意：后一种方法没有处理下面这样的代码序列：

```
int i;
Complex a, c;
a = i + c;
```

但如果 Complex 有一个接受单个 double 和 int 的构造函数，用重载的方法就可以处理此种代码序列。通常将重载结构（处理后一个问题）与超越结构（提供运行时属性）一起使用：

```
Complex operator+(const Complex &n1, const Complex &n2) {
    // would be declared a friend function in class Complex
    Complex retval = n1;
    retval += n2;    // a virtual function call
    return retval;
}

class Complex {
public:
    virtual Complex& operator+=(const Complex&) const;
    . . . .
};
```

甚至这种解决方案也有局限性。如果两个参数都是从 Complex 派生出来的类的对象，那么仅有 C++虚函数的单独调度模型是不够的（它无法容纳前面例子中 n2 的动态类型）而且，设计者必须采用诸如 **Visitor** 这样的模式。我们将在第 8.4.3 小节中讨论这一类的解决方案。

总之，对于通过一个共同的意义绑定在一起的函数族来说，函数参数的类型以及 C++语言的实用性驱动了对语言特性的选择。绑定时间还控制了对具体实现结构的选择。我们将在第 3 章中讲述多范型设计的这一重要方面。

2.4.3 算法

过程是最早的编程抽象之一，过程的解析是一种经久不衰的设计模式。我们要考虑两种类型的过程解析：自上而下和自下而上，每种方法都各有利弊。自上而下的设计允许我们设计出高层次的抽象，但它不能带来大量的代码复用，这样就使得高性能设计的实现变得困难。自下而上的设计更有可能创建出可以被广泛复用的基本算法，但创建高层次的抽象时就需要更多的工作量，这样就使得预见这样一个问题变得困难：最初的函数能够在多大程度上满足高层次的需求。

现在来考虑自下而上的设计中的共同性。我们要设计一个可以从上、下文中多个不同的点进行调用的过程。当然，考虑共享很多共同逻辑的算法族会更好。（注意：这里的共同性与第 2.3.2 小节中描述的行为的共同性是不同种类的。）过程编程的一个设计窍门就是将算法族打包到一个使用正式参数区分选择分支的过程中：

```
void FloorPlan::redraw(int showRoomDimensions = 0) {
    . . . .
    switch (showRoomDimensions) {
```

```
        case 2: // draw arrows annotated with room dimensions
            . . . .
            break;
        case 1: // just show square footage of room
            . . . .
            break;
        case 0: // no annotations at all
            break;
        }
        . . . .
    }

    . . . .
    FloorPlan *p;
    . . . .
    p->redraw(1);     // binding specified here in the source
```

或者我们可以分解出差异来，以便它们能在函数之外汇总：

```
void FloorPlan::redraw(void FloorPlan::*f() = 0) {
    . . . .
    if (f) this->f();
    . . . .
}

void FloorPlan::drawArrows() {
    . . . .
}

. . . .

FloorPlan p;
. . . .
p.redraw(&FloorPlan::drawArrows);  // source-time binding
```

这种方法将共同性封装在各个函数中。更进一步来讲，我们可以用一种更加面向对象的方式来达到同样的设计目标。

```
class FloorPlan {
public:
    // qualification is used just for emphasis
    void FloorPlan::redraw() {
        . . . .
        drawDimensions();
        . . . .
```

```
    }
    virtual void drawDimensions() = 0;
};

class FloorPlanWithArrows: public FloorPlan {
public:
    using FloorPlan::redraw;
    void drawDimensions() {
        // draw arrows for this floor plan
        . . . .
    }
};
. . . .

FloorPlanWithArrows p;
p.redraw();    // with arrows, compile-time bound
```

或者，我们可以用一种稍微不同的用法来达到这些目标，假设 FloorPlan 和 FloorPlanWithArrors 有相同的变量类型定义：

```
FloorPlan *q = new FloorPlanWithArrows;
q->redraw();   // with arrows, run-time bound
```

每种方法都捕获了算法中的共同性，也表达了带有不同绑定时间的不同变量类型的差异。

2.5 回顾共同性分析

在领域词典逐渐“演进”(evolve)、术语条目逐渐形成的时候，暂停下来思考一下共同性分析，这对我们是很有用的。词典的演进可能需要几个星期、几个月甚至更长的时间，所以设计团队应经常地回到这些术语上来思考它们。正如第 2.2.2 小节所述，一部分内省适合词汇表自身。我们可以将针对分析的这些问题扩展如下：

- 所有的族成员的共同属性是否都已经进行了陈述？
- 共同性是否可以更加精确地陈述出来？
- 这些共同性是否准确？
- 这些共同性是否一致？

在共同性分析和差异性（并行）分析（第 3 章）期间，在选择实现机制时（第 6 章），以及在维护期间发生设计变化时，我们都要努力回答这些问题。

2.6 共同性和演进

在共同性分析期间发现的抽象可以在最稳定的设计结构中得到表达，它们在实现结构中带有最强的惯性。它们包括继承层次结构、重载函数的族、模板例化的族。由于族的抽象之间、或者族与其客户端之间存在依赖性，所以一旦定义了这些抽象（它们就成了结构的基础），要改变它们就需要相当的代价。要重新安排继承图中的任何一部分，设计者就不得不重新回到针对所有从被移动的或被删除的类派生出来的类的设计中来决定。如果为重载函数族添加了一个新的签名，那么对必需的协同变化的作用域的评估将变得非常麻烦。如果某个模板参数发生了变化，则程序员必须重新编译模板本身以及应用中所有使用了此模板的代码。

如果要构建一个能够很好地适应未来发展的系统，那么这种系统必须能轻松地适应所发生的变化。要保证这一点就意味着：当变化发生时，一般不需要重塑或替换主要的系统结构。我们希望大部分的变化只产生局部的影响，不影响全局接口。简单地说，也就是尽可能使系统对这些变化“不可见”。反过来说，也就意味着我们想要让这些变化不影响实现共同性分析的抽象的结构。

所以在进行共同性分析时，我们要努力刻画出随时间保持稳定的抽象。这就意味着共同性分析要用到分析者对领域是如何随时间变化的了解。我们不仅要寻找领域词汇表中的结构之间的共同性，还要看明天的词汇表是否将与今天的一致。

我们可以将一个领域的稳定部分封装为共同性，但同时也不能忽略了领域中随时间变化的部分。在下一章中，我们将会看到差异性是如何管理跨应用或随时间不稳定的设计假设的。共同性分析和差异性分析都揭示了系统结构：共同性分析揭示了一般不随时间变化的结构；而差异性分析则捕获了可能改变的结构。差异性分析只在根据相关的共同性分析所定义的上下文中才有意义。但差异性分析捕获了系统演进方式。系统的维护可能更依赖于共同性分析中没有的内容，而不是依赖于共同性分析中所包含的内容。

幸运的是，共同性通常与时间的弹性密不可分。被称为“电话”的基本抽象在过去的一个世纪都保持不变，只有很少的一些“离群者”（比如可视电话）。因为电话的这些属性是普遍接受的电话行为（独立于实现）模型的本质，所以它们对于大多数电话来说都是共同的。这些抽象之所以能保持 100 年不变，也因为它们对电话需要做什么的基本关系的界定。电话的其他部分已经随着技术的改进而改变。但多数时候，一个好的共同性分析都预示着一个具有弹性的设计。

2.7 小结

本章探讨了共同性分析的重要性。共同性是我们在形成应用领域和方案结构（比如编程语言构建块和自定义类型）中的抽象时首先要寻求的东西。我们通过构建一个领域词典开始找寻共同性。这个词典突破单个领域来寻找复用和演进的预期中的应用族。领域词典捕获了结构、签名、类型和用例中的趋势。当代面向对象分析的共同抽象（类）是统一结构和签名中的共同性的二次的抽象。因为共同性分析突破了类，甚至还突破了类型和职责，所以它是一种比单独的面向对象的分析更加通用的方法。

分析的另一个方面是差异性。如果我们在共同性上停步不前，那么生活将变得很枯燥，没有任何变化。差异性是设计的一个有趣的部分，但它只在被很好地定义过的语境中才有意义。有了共同性和差异性，我们将可以审视应用领域和方案领域的模型，并了解如何为给定的问题将这两种结构结合在一起。

第 3 章
差异性分析

上一章我们讨论了共同性，它是构成大部分软件范型基础的两个重要维度中的第一个。这些共同性可以用一种语言（比如 C++）表达出来。本章我们将讨论范型的第二个维度：在一个范型中，类似的条目是如何区分的？族成员之间存在很多不同之处，但我们可以用参数来表示领域的差异性（也就是参数化领域的差异性），从而获得巨大的抽象能力。这里我们要着眼于共同性和差异性在 C++中的实现来讨论差异性分析。本章还将介绍差异性依赖关系图——领域结构的一种简单记法。

3.1 差异性：生活的调味剂

共同性分析是为了寻找一些共同的因素，以帮助我们理解“族成员是怎样相同的”这个问题。正如第 2 章所述，我们很善于发现共同性：我们注意到它，它吸引了我们的眼球。但如果没有差异性，共同性就会很单调。如果族成员之间没有差异，那么创建抽象来帮助理解族的整体就没有意义。所以在共同性形成设计的中枢和骨架的同时，差异性给予了其血肉。从架构的角度来看，共同性分析给予架构长期的寿命，而差异性分析驱动了架构的适用性。

第 2 章还讲到过，只有共同性分析是不足以指明、甚至暗示出一种解决方法的，尤其是当我们使用 C++作为方案领域时。例如，假设我们选择 **Text Editing Buffers**（文本编辑缓冲区）作为一个领域，因为它们的行为和大部分的结构都是共同的。（**Text Editing Buffers** 是本章中的一个连续示例。我们将根据对文本编辑器的适用性来讨论它们。因为它们被应用于很多文本处理应用程序中，所以也可称它们为 **Text Buffers**——文本缓冲

区。）如何表示共享共同的行为和结构的抽象？此时，类和继承映入脑海，尤其是对一个拥有“面向对象的思维”的人来说。但要选定一种实现策略，不仅要了解 **Text Editing Buffers** 是怎样相似的，还要了解它们是如何“不同”的。如果说文本缓冲区只在对加密的支持上有所不同，那么设计者就可以用#ifdefs 取出每一种加密的算法。如果它们在所支持的字符集上有所不同，那么一个好的 C++程序员将研究把模板作为一种解决方案。如果它们使用了不同的存储策略和算法，那么继承将是一种可行的解决方案。

应用领域结构可以沿着用来判断共同性的同一维度面变化。尽管 **Text Editing Buffers** 共享共同的行为，但可能在算法、详细的类型特征，以及它们的成员函数所接受的参数的类型上存在不同。这些差异都将导致对不同 C++技术的选择：根据不同算法选择继承，根据不同类型参数选择模板，或许还要根据不同成员函数参数类型选择重载。

即使理解了差异性维度，我们还必须理解绑定时间和默认值。**Text Editing Buffers** 可能包含针对两种不同的内存管理策略的源代码。所有的这些代码是否都出现在给定的编辑器的运行副本中，以便在运行时间我们可以在两种内存管理策略之间进行选择？或在构建此程序时，我们是否需要先用#ifdefs 来捆绑决定，将其中的一个实现编译到程序中？或在以上两种情况下，哪一种缓存管理策略应作为默认值，以便初级用户在没有选择任何选项时遇到“合理的”行为？

我们将在差异性分析期间探讨所有的这些问题。因为差异性分析和共同性分析都是用了相同的共同性类别的分类方法，所以这两种活动应当在一起进行。领域工程用捕获共同性、差异性、绑定时间、默认值和领域之间关系的符号来支持共同性和差异性分析。共同性和差异性分析的“输出”就是这样一个符号集。通常，在进行差异性分析时，我们可以将这些符号作为各种层次设计的直接输入（编写类和函数的原型）。本章介绍差异性的维度和简单的差异性符号。

3.2 共同性基准

差异性只在给定的参考共同性框架中有意义。在分析差异性时，很重要的一点就是要保持“某些事情”固定不变。共同性分析有两个用途：找到系统中结构的主要维度，并为差异性分析提供一个背景。差异性及其共同性基准之间的关系是已发布的差异性分析的重要组成部分。

我们不能脱离语境来讨论差异性，但肯定会不停地问：“差异性在哪里？”我们来考虑一个 Message（消息）。当我们说消息的长度不同时，隐含的共同性就包括使消息之所以成为消息的特性。更进一步地审视，我们会发现这些特性包括给定格式和长度的消

息头、给定格式和长度的消息尾，以及不同长度和未指定格式的消息体。这里的共同性是每个消息都包含消息头、消息尾和消息体。对于每一种消息类型，消息头和消息尾的格式是固定的。理论上我们在共同性分析期间找到这些属性。消息也还有很多其他的共同性，例如，用于将它们发送到网络介质上或计算它们的校验和的算法。

共同性本身常常为差异性提供了关键的提示。我们来考虑下面这个共同性：

所有的 **Text Editing Buffers** 都有一些类型的工作区算法。

千万别搞错了，这是共同性。但它也为差异性指明了道路：**Text Editing Buffers** 所支持的工作区算法的种类不同。大部分情况下，我们感兴趣的差异性就在这个类别中。因为共同性和差异性分析之间存在这种紧密的联系，所以两者在一起进行，而不是分离的、有时间先后顺序的活动。

3.3 积极和消极差异性

我们重新考虑上一节中 **Message** 的例子。所有的 **Message** 都有给定格式的消息头和消息尾，我们注意到这是 **Message** 领域的一个重要共同性。尽管“消息体”成员的存在是 **Message** 领域的一个基本的共同性，但领域成员之间的消息体存在很大的不同（我们暂时忽略没有消息体的消息）。我们可以统计拥有一个消息头、一个消息体和一个消息尾的所有消息的数量。注意：消息头的格式对特定的系统是固定的，但跨系统和跨时间的可能不同。

3.3.1 积极差异性

消息长度的差异性和头字段内容是独立于使消息之所以成为消息的共同性的。一个消息可以保留某个头字段为空，此时它仍然是一个消息。也就是说，它可以有 1 个字节或 256 个字节的消息体，并且它仍然作为一个消息处理和传送。但头字段的格式和消息体的长度也是重要的差异性。因为这些差异性没有触及作为基础的共同性模型，所以它们向消息“添加”一些东西，以改进这些消息的定义。我们将这些称为“积极差异性”（positive variability），因为它们向基准规格说明（这些规格说明是消息的本质特性）列表中添加一部分内容，而没有违背任何原始的规格说明。

如果差异参数是一个积极的差异性，那么当我们用一个值替代这个参数时就会得到一个弱抽象族。当我们捆绑越来越多的差异参数时，我们就创建了族成员集合的不断增长的约束性规格说明。例如，**Text Editing Buffers** 在一个抽象层次上描述了一个大型的族。特殊的编辑缓冲区可能支持特定的“字符集”（character set），这里的 **Character Set**

是差异参数之一。因为当我们将 **Character Set** 捆绑到一个特定的选择时带来了更具约束性的规格说明，所以 **Character Set** 是一个积极差异性。它是积极差异性还因为它的绑定没有违反 **Text Editing Buffers** 领域的共同性。在捆绑了所有的差异参数之后，自由程度达到了最小值，于是产生了单个族成员。

3.3.2 消极差异性

如果所有的消息都有消息体，那么差异性被限制在消息体的长度上（假设消息设计并不关心消息体的格式）。如果多数的消息都有消息体，而有一部分没有消息体（比如 ACK 消息——“命令正确应答”消息），那么差异性就打破了共同性假设。我们称这种差异性为“消极差异性”（negative variability），消极差异性中的差异性与我们定义“消息”意义的假设相矛盾。相对于积极差异性来说，它是一种根本不同的差异性类型。

有人可能会这样认为：所有的消息都有消息体，类似 ACK 这样的消息有一个 0 字节长度的消息体。如果采用这种观点，我们就从设计中删除了消极差异性。但这实际上扭曲了 ACK 消息的语义，至少它违背了常识。这种设计重铸可能增加成本和复杂性。它带来了一些麻烦的问题，比如“可否获得 ACK 消息的消息体的校验和？”如果可以，那么 ACK 消息就承受校验和代码的开销。这将使类 `ACKmessage` 的接口变得复杂，此接口中出现的校验和是语义开销。依据涉及到的共同性和差异性的类型，这可能导致实际内存和实时开销。我们应当在逐个用例的基础上处理这些设计决定，因为有时它们确实存在意义。例如，所有的 `Shape` 都可以报告各自的旋转角度，尽管我们能够安全地为 `Circle` 去除这些属性。我们通过使这种属性普遍存在而获得的一致性，超过了通过区别对待 `Circle` 所获得的空间和语义的轻微缩小。

消极差异性存在一个范围。多范型设计提供了创造性，但却为处理“小的”消极差异性提供了直接的方法。例如，为了处理“缺少某些消息字段”，可以根据消息类型使用 `#ifdef` 来选择代码：

```
// adapted from stropts.h in a UNIX header file

struct strrecvfd {
#ifdef KERNEL
    union {
        struct file *fp;
        int fd;
    };
#else
    int fd;
```

```
#endif
    unsigned short uid;
    unsigned short gid;
    char fill[8];
};
```

假设 KERNEL 选项是主流的共同基准。那么非 KERNEL 版本则提出了一个消极差异性，因为它从假设的结构共同性中去除了 fp 成员。如果假设非 KERNEL 代码是主要的共同基准，那么 KERNEL 版本则去除了 fd 成员作为一个字段（可以确信其值有效）的首要位置。换句话说，差异性损害了假定的共同性，所以它是消极差异性。

这个例子可能无法很好地评定或推广。如果 strrecvfd 消息只有一半用在了内核中而另一半没有用，那么就很难说 fp 是 **Strrecvfd** 共同性领域的一个基本构件。我们可以选择将此结构分成两个单独的领域：**Strrecvfd** 可能用于文件描述，而 **Strrecvfp** 用于文件指针。这种分割就使消极差异性不存在了。

我们可以跨多个领域并跨多个差异参数应用这些积极和消极差异性的方法。这些共同性和差异性的方法已针对对象范型下的类继承进行了充分地开发。正如 Synder[Snyder1986]的半年度分析中分析的那样，带相消的继承是消极差异性的一种特殊形式，带加法的公共继承是积极差异性的特殊形式。

在使用多范型设计时，应尽可能试着使用积极差异性来表达设计。通常，多范型常可以使用已存在的方法（比如带相消的继承）来容纳消极差异性。最直接的方法是将这种领域沿着消极差异性这条线分成不相交的多个子领域。从#ifdef 的明智使用到更加通用的解决方案中产生的更多的高级方法将在第 6.11 节和本书的其他地方讲述。

3.4 差异性的领域和范围

差异性区分了软件族的各个成员。区分族成员的参数的数量可多可少。理解区分族成员的差异参数是很重要的（有多少差异参数，以及它们带有什么样的值）并要理解适当的参数值是如何满足设计需要的。

本节将介绍描述族成员之间的关系以及族成员与形成它们的方案之间关系的词汇表。我们将一个族看成是一个成员集，这些成员可由满足特定设计需求的机器生产出来。我们用术语“范围”(range)来表示机器可以生产的族成员的整个范围。这种机器有输入、按钮、开关和控制我们要生产的族成员属性的杠杆。我们称这些输入为“差异参数”（parameter of variation）。差异参数的数量可多可少，每个参数可接受几个值或很多值。我们使用术语“领域”来表示针对一个族的所有差异参数或针对一个特定的差异参数的

值的有效结合。注意：这里使用的“领域”和“范围”与它们的数学意义基本相同。领域工程的首要任务就是要通过发现共同性、差异性并描述此领域的差异参数，从而建立这样一个“机器”。

共同性帮助我们进行抽象，要获得抽象就不能强调细节。我们将差异性看作是对共同性的补充，它们表达了非共同的东西。那么，这是否意味着差异参数表达了细节？有时，差异性比共同性显著，这时“差异”就凸现出来。在共同性分析期间我们仍然要捕获主要的共同性，这对复用有帮助。但我们又不想因此而淡化了差异性的重要性。相反，我们要使它变得有规律。如果我们可以将很多差异性抽象到等价集合中去，那么我们就有可能依据一些简单的参数来描述所有的差异性。这种抽象方法与我们从族成员之间的共同性中形成的软件族的抽象是不同的。可以牵强地这样类比，就是大自然是如何管理共享同一个基因结构的物种个体之间的差异性的。

3.4.1 Text Editing Buffers 例子

所有的差异性分析都在一个已建立的共同性领域的语境中进行。考虑第 3.2 节中文本编辑器的 **Text Editing Buffers**（文本编辑缓冲区）领域。**Text Editing Buffers** 根据诸如结构和算法这样的共同性形成了一个族。不同的缓冲区拥有不同的输出类型、字符类型和工作区管理算法的范围。输出类型、字符类型和工作区管理算法都是差异参数。正是这些参数控制了族中的差异性的范围。

每个差异参数都有一个相应的值“领域”，它是一个不依赖于设计变量（我们选用其值生成共同族的一个特殊成员）的集合。领域是值的一个集合，它描述相关范围中差异性程度的特征。对于 **Text Editing Buffers** 来说，控制输出类型差异性的领域包括数据库、RCS 文件、屏幕输出和 UNIX 文件。字符集属性是通过从 ASCII、UNICODE、EBCDIC 和 FIELDATA[①]集合中选择来进行控制的。

差异性领域集合是设计的“加压点”。通过将适当的领域集合成员赋值给差异参数，我们就控制了软件设计的族成员的生成。在一个好的设计中，编程语言捕获了背景共同性，但它强调表达差异参数的方式。

例如，我们可能非常了解 **Text Editing Buffers** 的复杂性，并要在共同性分析期间探究并捕获这种复杂性。编程语言应当隐藏所有 **Text Editing Buffers** 所共享的这种复杂性。我们应使用语言设备——继承、`#ifdefs`、重载、模板等——来控制输出类型、字符集和特定文本编辑配置中使用的工作区算法。我们将在第 6 章中分析这些疑难语段，并在

① 根据作者在 20 世纪 70 年代的经验，这是用于 UNIVAC 机器上的 6 位字符集，现已作废。

第 8 章中将差异性分析与语言设备结合起来。

3.4.2 好的差异参数

好的差异参数从何而来？在设计进行过程中，设计者可以问自己下面这些关于差异参数的问题：

- 参数领域是否覆盖了应用领域差异性的范围？如果设计要支持多版本发布，并且公司的多种产品都要用到此设计，那么选择好的差异参数就至关重要。差异参数对领域的宽度进行了编码。
- 有没有无法追踪差异性的差异参数？或反之，是否存在没有被差异参数追踪的差异性？
- 默认值是否合适并在参数领域中？（默认值将在第 3.6 节中详细讨论。）

3.5 绑定时间

分析主要是一项“获得关联”的工作；共同性分析要寻找构建块的应用族的共同关联。关联可以通过应用领域、产品的使用、产品的实现形成。给定产品可能需要服务于多个差异性配置，这就意味着我们可能需要在给定产品中容纳很多选项。

3.5.1 绑定时间和灵活性

当我们需要为多种产品选项预见需求时，我们可以将软件族的成员之间的选择推迟到什么时候？编译时间、运行时间、还是介于两者之间的某个时间？用软件经验法则捆绑一个决定是越迟越好，因为后来的绑定和灵活性是密不可分的（正如第 1.8.3 小节中所述）。当然，我们需要用非结构的（通常也是非功能性的）工程考虑事项（比如性能和静态类型检查）来平衡这种经验法则。但如果灵活性对我们很重要，并且推迟绑定时间不会引起损失，那么我们就要尽可能地推迟绑定。

除了共同性和差异性以外，通常我们还必须理解绑定时间，从而为给定的分析选择一个合适的实现方法。多范型分析使绑定时间很明显。绑定时间选项是可用的解决方案方法的一个函数。这些选项通常比单个“编译时间——运行时间”模型丰富，所以通常在对象范型的讨论中描述，我们将在第 3.5.4 小节中讨论它们。

3.5.2 对象与推迟的绑定有关吗

在很多对象范型的教学中，推迟的绑定都是仔细教授的课程之一。在面向对象的设计和

编程中，我们通常将对给定类的选择一直推迟到运行时间，然后再从语境中进行选择。面向对象的编程语言将这种灵活性与隐藏决定于基类接口下的能力结合在一起。继承是隐藏差异性的方法之一。虚函数提供了运行时间绑定，并完善了支持透明差异性所必须的机制。继承和虚函数提供了强大的运行时间差异性，以抵抗基类结构和接口的共同性的干扰。

3.5.3 效率和绑定时间

我们要为运行时间绑定付出代价。通常，一个程序必须包含针对所有可能运行时间选项的代码。如果我们对每次函数调用都使用运行时间绑定，那么所装载的每个程序都将包含全部代码的完整拷贝，而不论我们是否期望用到它。而且，这些程序都需要包含某种逻辑，以便在语境需要的时候打开或关闭某个选项。这种逻辑妨碍了对程序的理解，并牺牲了程序的性能。并且因为弱化了静态程序检查，而同时我们希望有一个完整的编译时间的类型检查，所以我们总对缩编的程序缺乏自信。当然，我们很少走向这种极端。相反，我们经常在运行时捆绑差异性的一些维度。（动态链接库是个奇特的例外，但相对于设计的一个重要维度，将它看作是一个程序工程方法可能更好。）

3.5.4 绑定时间的选择

多数设计者都考虑两种绑定时间的选择："编译时间"（compile-time）和"运行时间"（run-time）。我们可以考虑得更加精确一些，尤其是对于那些将 C++作为实现领域的设计。绑定时间包括"源码时间"（source time）、"编译时间"、"连接时间"（link time）和"运行时间"。

源码时间

源码时间绑定意味着程序员明确地从程序源代码中选择合适的族成员。模板参数就是一个例子：

```
List<char>
```

这里程序员显式地替换了差异参数。

编译时间

编译时间绑定意味着编译器可以在编译时从语境中选择适当的绑定。C++函数重载是一个典型的例子——程序员没有明确规定差异参数，但编译器对其进行了规定。一个特殊的例子就是`#ifdef`。尽管选择结构在源代码中，但差异参数可以在相隔的头文件中，或者可以作为一个编译器标记来提供。编译时间绑定非正式地暗含了源码时间绑定。

连接（装载）时间

连接时间绑定意味着可以在程序运行之前选择预编译代码来满足某个接口，这通常

要依赖于环境，但 C++对此并不支持，所以在我们的分类方法中没有包括此项。要了解有关 C++程序的连接编辑和装载时间的技巧的更多内容，请参考 Coplien[Coplien1992]。

运行时间

虚函数是运行时间绑定的一个典型例子。编译器在编译时间生成代码以处理运行时间的选择，它无法预知运行时间的语境是什么样子。

3.5.5 一个例子

我们再来考虑 **Text Editing Buffers** 的例子。通过从相同的基类派生出多种实现，我们可以使编辑器完全通用，以便它可以在运行时处理任意的字符集，如图 3.1 所示。

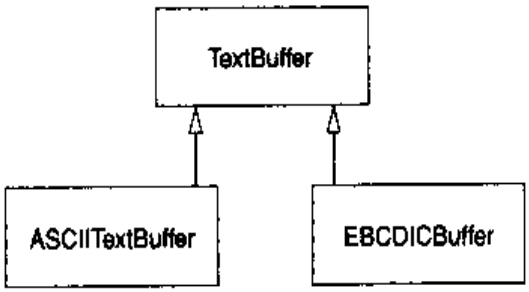


图 3.1 文本缓冲区的一个完全通用的实现

但这样每个编辑器都将包含针对 `ASCIITextBuffer` 和 `EBCDICBuffer` 的代码的完整拷贝。给定版本的编辑器在某一时刻只处理一个字符集，并要依赖于编译和连接的目标平台。我们可以使用模板，以便能够连续捕获源代码中的差异性，同时无需记录实现中完全补充的选择，如图 3.2 所示。两种设计都捕获了共同性和差异性分析的相同模式，但它们的实现绑定时间不同。

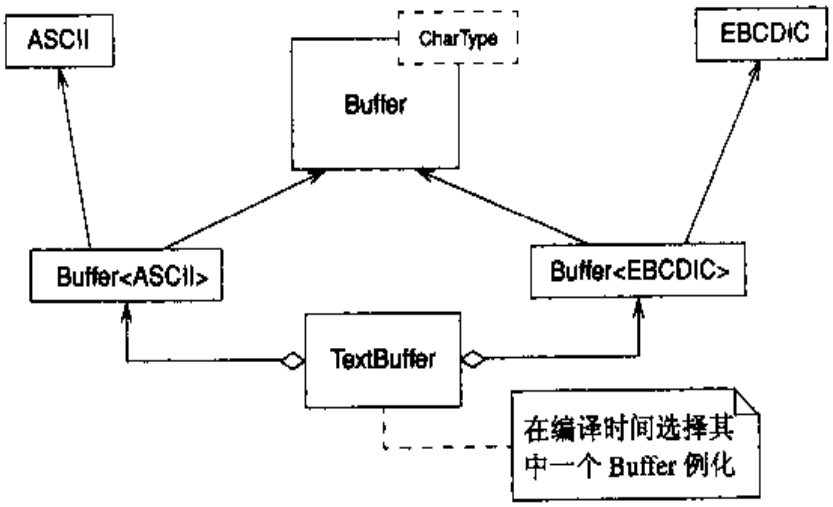


图 3.2 提供了实现中的灵活性的通用代码结构

在进行差异性分析时，我们必须为每个差异参数捕获绑定时间，因为它将指导我们在捕获了设计意图的合适的编程语言特性中进行选择。

3.6 默认值

当差异参数对应用的影响很小时，默认值就是很有用的方便措施。C++拥有表达默认值的有用设备，如参数默认、基类成员函数和默认模板参数。我们在共同性和差异性分析期间所捕获的默认值经常可以直接用编程语言来表达。

大多数人都很熟悉将#ifdefs 作为捕获差异性的方法，也很熟悉如何使用它们来区分不同的行为和代码结构。或许我们想要在文本缓冲区的实现中放入调试代码，但又不想将此调试代码包括在目标代码的产品版本中。这样，就可以将调试代码设置为默认值：

```
#ifndef NDEBUG
    // include this code by default
    cerr << "got to " << __FILE__ << __LINE__ << endl;
#endif
```

如果使用#define，可以生成一个产品版本。如果产品版本比调试版本的频率小得多，则将调试作为默认情况是比较合适的，这将为开发者减少工作量。

设计者还要记住：默认值是很危险的。一个复杂的设计可能有多个差异参数，每个差异参数都要进行仔细的考察，看它是否支特给定的部署语境。对设计者而言，将默认值遗漏到实现中而无法更改是一种很冒失的行为。默认值应当仔细地进行选择（如果它们存在），并应当作为工程架构文档的一部分明确地记录下来，还要针对每个应用进行相应的修改。

3.7 差异性表

我们可以生成一个差异性表来记录我们的差异性分析的结果。每个共同性领域对应于一个表，每个应用可以包含多个共同性领域。

表 3.1 显示了对 **Text Editing Buffers** 进行差异性分析的结果。**Text Editing Buffers** 是整个文本编辑程序设计的一个主要语境中的共同性领域。我们在针对结构和算法的共同性进行的共同性分析期间认识了 **Text Editing Buffers** 领域。那就是所有的 **Text Editing Buffers** 有何共同之处，差异性表描述了它们是如何相异的。

表 3.1 共同性领域 Text Editing Buffers 的文本编辑器差异性分析

差异参数	含义（产生的设计决定）	领　域	绑　定	默认值
输出类型	输出介质对文本行的格式敏感	数据库、RCS 文件、TTY 和 UNIX 文件	运行时间	UNIX 文件
字符集	不同缓冲区类型应支持不同的字符集	ASCII、EBCDIC FIELDATA、UNICODE	编译时间	ASCII
工作区管理	不同的应用需要在内存中缓冲不同数量的文件	整个文件、整个页面 固定 LRU	编译时间	整个文件
调试代码	调试陷阱是为了内部开发而出现，并永久性存在于源代码中	调试、产品	编译时间	无

我们看到：跨输出类型、字符集、工作区管理方法和调试代码配置存在一个 **Text Editing Buffers** 范围。表中的各个列对每个范围、范围的绑定时间和默认值（如果存在）进行了精化。附加列提供了对差异参数（领域变量）的简单描述。

第 8 章中将同时使用这些表格和 C++语言结构的模型，以获得解决方案的结构。

3.8 一些差异性陷阱

我们来考虑 **Text Editing Buffer** 的差异性表的条目：

差异参数	含义（产生的设计决定）	领　域	绑　定	默认值
窗口类型	文本缓冲区写入很多不同类型的窗口	X、Curses、Ms Windows	运行时间	无

这里，我们没有将窗口当作与表 3.1 所示的 **Output Type** 子领域类似的输出介质，而是将它们当成同一个系统中一个独立的、自由的耦合领域。设计者已经将 **Window** 类型指定为一个差异参数。为什么要这样指定？因为不同的 **Text Editing Buffers** 共同存在于拥有不同类型窗口的系统中。这些类之间不太可能进行交互。可能的情况是：一个类（比如 `EditingLanguage`，类似与一个 MVC 控制器）将内容从缓冲区发送到窗口。即使

Window 和 **Text Editing Buffer** 领域进行交互，窗口类型也不会改变 **Text Editing Buffer** 的实现。给定的 **Text Editing Buffer** 可以与任何类型的 **Window** 共同存在。两者可以独立地进行设计。

只就两个领域之间存在关系这一点，还不能说明其中一个领域就是另一个领域的“参数”。我们还需要确认：所有的领域都对所列出的差异参数敏感。

这里存在另一个常见的缺陷，我们用 Paul Chisholm 推荐的一个例子来说明。考虑 `string` 的差异性表，如表 3.2 所示。此分析将带来这样一个设计，其中诸如 `PathName` 和 `RegularExpression` 这样的类都是从 `string` 派生而来。我们将差异性分解成派生类分析函数，此函数超越了一个基类虚函数。正确的设计是将诸如 `PathName` 这样的类从 `string` 中分离开来。这是因为 `PathName` 和 `RegularExpression` 的有些属性违反了所有 `strings` 的共同性（参见[Coplien1992]，第 6.5 节）。例如，我们不能用一个任意的字符（比如“/”或“*”）来覆盖任意一个 `PathName` 字符（与 Modula 3 中的路径名抽象形成对照）。

表 3.2 String 的差异性表

差异参数	含义（产生的设计决定）	领　域	绑　定	默认值
分析算法	不同类型的字符串应当进行不同的分析；它们服从不同的词法	Character string、PathName、Regular Expression	运行时间	无

注意理解族的共同性和差异性，并注意所有所谓的族成员都展示了所声明的共同性，并可以用已知的差异参数表示出来。

3.9 回顾差异性分析

正如第 2.5 节所述，对共同性分析的结果进行内省是很重要的。同样，回顾差异性分析的结果也很重要。因为差异性和共同性都是只有在彼此的语境中才有意义，所以在实践中我们同时来回顾二者。我们可以从差异性的角度对分析提出以下几个问题：

- 差异性是否涵盖了全部的族成员？
- 每种差异性是否可以描述得更加精确？
- 是否存在不一致的差异性？

- 差异性是否能很好地预测变化（领域分析）？
- 排除了什么样的族成员（也就是说，不能指定什么样的族成员）？

3.10 差异性依赖关系图

这里，我们将介绍一种常用符号表示法，它可以使用简单的图来显示领域与它们的差异参数之间关系。我们称这些图为“差异性依赖关系图”（variability dependency graphs）。本书后面的部分（第 7、8 章）将使用这种符号表示法来组织软件架构。

现在来考虑用于收集和加密文本集的领域 **EncryptionBuffer**。可以设想一个使用不同加密算法的缓冲区族。**EncryptionBuffer** 族的特定成员使用特定加密算法。在我们的简单符号表示法中，我们将 **EncryptionBuffer** 描述为一个带有指向差异参数箭头的节点，如图 3.3 所示。

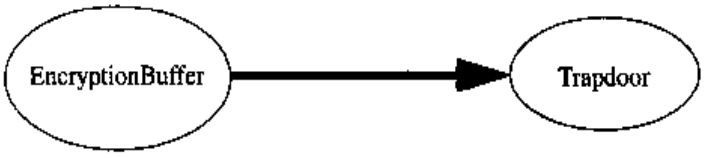


图 3.3 **EncryptionBuffer** 与 **Trapdoor** 之间的领域依赖关系

就其本身而言这并没有什么意义，但如果我们看得更深远一些，这种表示方法就很有意义了。首先，**EncryptionBuffer** 也可以把 **CharacterSet** 作为差异参数。不同的加密缓冲区需要针对 **UNICODE** 而不是 ASCII 的不同内部存储结构。更有意义的是 **Trapdoor**，我们将其看作一个差异参数，实际上它自身可以看作是一个领域。它的代码也依赖于 **CharacterSet**，这就意味着：作为一个领域，它也将 **CharacterSet** 作为一个差异参数。所以我们可以进一步完善图 3.3 来捕获这些语义，如图 3.4 所示。

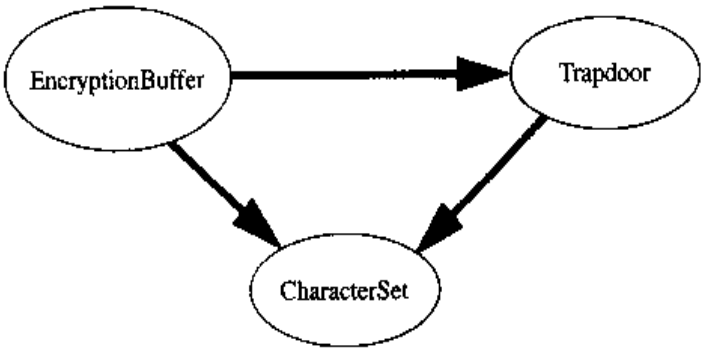


图 3.4 在图 3.3 中加入 **CharacterSet** 依赖关系

这些图可用来映射出设计中的领域之间的关系。通过差异参数连接扩展的关系链可能非常复杂。这种简单的符号表示法帮助设计者理解整体结构是什么样子、给定差异参数变化所带来的影响如何，以及领域是如何相互依赖的。在第 7 章和第 8 章中我们将会看到，这种简单的符号表示法还有其他很多的功能强大用法。

3.11 小结

本章介绍了一个差异性模型，用于描述有关软件抽象之间的不同。差异性是软件族的通用代码的关键。共同性模型与差异性模型之间存在密切的相互关系，并且我们必须仔细地鉴别积极差异性和消极差异性。如果可能，应当对消极差异性进行从单个领域分离为多个子领域的处理。如果无法做到这一点，则必须退一步考虑诸如`#ifdefs`这样的解决方案。

我们学习了差异性的领域和范围、差异性的绑定时间及其默认值。这些都是差异性分析的主要描述特性。我们可以在差异性分析表中捕获这些属性，并使用此表将分析结构与表达它们的 C++编码结构进行匹配。

第 4 章
应用领域分析

本章介绍领域分析。在进行传统的分析时，我们研究的是一个特定的问题或领域。在进行领域分析时，我们研究的是一个问题族或应用族。我们将使用这些领域中的共同性和差异性来描述族成员的特性。

4.1 分析、领域分析和其他

分析是对一个问题的仔细研究。“领域分析”（domian analysis）是对相关问题领域的仔细研究。多数设计方法都强调应用领域分析，通常称为“问题领域分析”（problem domain analysis）。我们使用术语“领域分析”是为了强调：不应将所有的软件工程都看作是一个问题，而应将它们看作是多种变化的焦点。分析不仅为应用解决问题和工程的方法提供了机会，而且它还具有创造性和艺术性。在看到本章中出现的方法、符号表示法和过程时，要记住：创造性是设计的核心。“抽象在某种程度上是一种艺术”，理解这一点很重要。所有的设计者都是运用他们的经验、学识和观点来抽象出大型或复杂问题的本质属性的。

本书将领域分析看作领域工程更广泛涉及的一部分。工程规范不仅聚焦于眼前的问题，还聚焦于方案领域的用语和形式方法。任何正式的分析都只有在期望实现时才有效。多范型设计通过使用一个共同的形式方法集合（称为“共同性类别”）将应用分析与解决方案紧密地联系在一起，以描述设计的这两个侧重面的特性。尽管本章聚焦在应用分析上，但本章将使用预期可用解决方案策略的符号表示法和词汇表。在第 6 章中，我们将跨出应用领域，考察“方案领域分析”。这两种分析在一起为 C++领域工程中领域层次的

问题提供了领域层次的解决方案。

4.1.1 传统分析

大部分受过正式训练的软件工程师都学习过分析艺术。学习过程中我们需要分析的“事物”通常是需求文档，而结果则是称为“架构”的东西，或是一个“高级设计文档”。每种情况中的术语都定义得很宽松。其他人则将这种“事物”看作是从非正式的用户需求到需求文档的转换。事实上，我们学会了将分析看作一门艺术，而不是将其看作计算机科学的一个组件。

我们要学习分析的需求文档是很复杂的，常常意义模糊，并且几乎总是不完整，尽管它们通常代表了单个客户对单个应用的需求。正如第 2 章所讨论的那样，抽象是一个关键的设计和分析工具，它可以帮助我们管理大型的或复杂系统的复杂细节，或将它们推迟到以后处理。第 2 章讲述了共同性分析是如何帮助我们形成抽象“立体型”（stereotype）[Wirfs-Brock1990]和“架构型”的，二者可用于对系统进行分析。共同性分析还帮助我们发现一个应用中类似抽象的族。帮我们识别可被系统其他部分使用的代码，给定软件族的所有成员可以共享代码。这样，好的分析就为软件复用提供了一个重要的基础。

我们偶尔也会在对族成员的分析期间发现抽象；在单个系统中找到最简单的软件族种类。我们可以类推出建筑架构：任何给定的建筑的大部分窗口都共享共同的属性，同时多个建筑之间的窗口可能存在广泛的差别。在软件“片断”中，我们可以“发现”输入/输出（I/O）设备族、消息族、数据结构族、行为族，它们都在单个需求文档所描述的单个应用中。一个勤奋的设计者可以通过构建领域词典（第 2.2 节）并与客户组织一起开展领域分析来获得对这些族的深入了解。

4.1.2 系统族：领域分析

在多数实际的软件开发项目中，开发者必须一开始就考虑多种部署配置。甚至开始是单个系统的工程，也会随着市场的增长和客户需求的扩展而分成离散的专用产品。只分析眼前的应用是不够的，我们要将视野扩展到整个应用领域、扩展到我们的市场领域、或者扩展到公司全景领域。我们要进行“领域分析”，而不是“应用分析”。

什么是领域？美国传统词典将其定义为“活动、关注或功能的范围，比如：历史领域”。许多人想要通过领域分析研究基本的业务抽象。金融贸易是一个领域，其抽象包括交易、股票、债券、证券、期货、衍生和外汇。音频通信是一个领域，其主要的抽象包括呼叫、电话、线路、干线、用户、特性。这些都是应用领域。每个领域都定义了一个

业务、公司或市场，我们在其中可以找到完整系统的族。

在数学中、在领域和函数的范围的意义上，术语“领域”也可以有更正式的含义。在第 3.4 节中我们曾间接提到“领域”的这种含义。在多范型设计中，术语“领域”同时表现了这两种含义。某种程度上，这种含义的定位支持了我们用方案领域语言的正式结构来表达应用领域的重要抽象。

本书介绍了围绕着领域概念而建立的设计和实现方法。领域分析是识别软件族的方法的集合，而应用工程则是实现和管理软件族的方法的集合。领域分析和应用工程在一起形成了我们称为“领域工程”的规范。多范型设计是领域工程的一种形式，为了实现领域工程，我们将领域分析应用于应用领域和方案领域。

在第 4.2 节中，我们将会看到：这些广泛的领域都是从更小的领域（子领域）编织而成的。GUI 是一个子领域，其抽象包括窗口、菜单、对话框、文本、字体和颜色。在面向对象的分析中这些抽象常作为类出现，尽管它们中的许多都因为规模太大，而不适合成为一个简单而内聚的好类。Booch[Booch1994]使用术语“类类别”（class category）来描述给定子领域的类层次和关联的类。类类别对面向对象的分析和设计中所形成的抽象进行分组。在多范型设计中，我们想要应用推广到对象范型所提供的标准以外的划分标准。

这比传统分析中所遇到的抽象的作用域更广。我们不仅要在某个应用中，还要跨越有关的应用来寻找共同性。应用系统本身形成了族。我们要讨论通过共同特性和属性联系起来的、以及针对具体市场而具有不同功用的市场生产线。一个好的生产线架构会将共同的部分从可变的部分中分离出来。好的设计会封装共同的属性，并识别针对具体市场而表现出差异性的架构的压力点。设计必须产生一个实现（用实际的代码表示），通过适当地绑定差异参数，它可以方便地对每一个市场段进行配置。

应用领域和方案领域中的族成员

某些差异性控制着算法和数据结构中底层的细节。许多重要的“市场”差异性影响了系统“实现”的细微结构，同时却保持总的结构不变。当然，通过包含或排除具体的模块，许多市场依赖关系可以直接在较高层次中表达出来。但一般而言，应用差异性的重要性可以不对应于与其实现的作用域或抽象的层次。

例如，我们考虑这样一个文本编辑器，它被设计为可处理几种字符集中的任何一种。（举这个编辑器的例子是为了说明本章和下一章的许多问题。）可以存在一个名为 **Text Buffers** 的领域，它关注的范围是给定时刻被编辑文本的工作区。此领域可以支持所有文本编辑器共同拥有的基本编辑功能，也可以参与到磁盘文件或来自磁盘文件的输入和输出中，以及到屏幕的输出中。此领域的多数“目标”代码都依据 `Chars`（而非 `wchar_t`）

字符而不同。这些不同之处并不明显，零星地分布在整个目标代码中。通过在源代码层次（为 **Text buffers** 类和函数）指定一个模板参数，我们可以控制针对特定市场的目标代码的生成。模板提供了一种便利的、固定的机制来表达这种市场差异性。模板参数可以很轻易地传给所有的 Text buffers 代码，此时模板参数还可以对目标代码的生成进行微调，以适应具体的市场。

因为应用领域分析可以在多个间隔的层次上影响解决方案的结构，所以一前一后地处理问题领域分析和方案领域分析是很重要的。正因为同样的原因，通盘考虑领域的所有的族成员也是很重要的。领域分析是跨整个领域的抽象的共同性和差异性的综合视图。尽管相对于传统的分析，它可以在一个稍高的层次上进行，但不应当将领域分析看作是将多种分析联系在一起的一个层。单个领域分析同时跨越了族中的所有系统。

平衡抽象和规格说明

领域分析比传统分析更加抽象。抽象好就好在它方便了复杂领域的导航：抽象的层次越高，复杂性就越低。Weinberg[Weinberg1971]指出：有时解决一个通用的问题比解决一个具体问题要容易。而且，尽管针对具体用例的一个直接解决方案可能更难以获得，但通用解决方案可以“推广”到具体用例。抽象的层次越高，抽象可以应用的范围也就越广。正是适当的文档和经济学所支持的这种抽象为复用提供了基础。来自于领域分析的抽象适用于领域中的所用应用，而来自于传统分析的抽象只能保证适用于眼前的应用。

一个好的领域分析会使用规格说明对抽象进行平衡。抽象隐匿了细节——这让我们丢失了一些设计信息。抽象的层次越高，设计所传达的信息就越少。抽象的层次应当与业务或企业的作用范围相对应。如果抽象扩展到领域的预期业务应用之外，或者扩展到其某个抽象之外，都可能导致丢弃了对领域很重要的一些细节。另一方面，如果抽象没有覆盖到业务领域，则进化可能将系统带到架构边缘以外，这将导致不明智的和高成本的架构修改。领域设计者必须对它们的业务有深入的理解。

例如，我们来考虑一个帮助公司管理联邦公司所得税减免的软件包。引人关注的一些子领域将是金融票据、销售、购买和投资。好的设计者会根据客户需求和他们在“联邦”（Federal）税法领域中的专门知识，在每一个子领域中寻找差异参数。这可以为美国市场带来一个优秀的、可管理的产品。但如果公司想要扩展其产品以处理“州”（State）税问题，它将会发现在“联邦”层次上没有出现的领域。如果公司想要扩展其产品以处理“国际”税问题，事情就变得更加糟糕。将最初的问题看作一个“可征税性管理”，这本身就忽视了捕获市场信息（市场所关注的内容）的潜在差异参数。因为设计者没有充分了解这个领域（可能他们忽视了“联邦”和“州”纳税义务之间的不同），或者没有充分了解业务远景（它们将自己过早地局限在“联邦”税市场，认为永远都不用关注“州”

或“国际”税市场），所以才陷入了这种“陷阱”。

回到编辑器的例子上，我们来考虑一个通用编辑器，要求它可以跨多个领域获得复用。我们可以让这个编辑器满足“通用”（general-purpose）的要求，从而使它能够适用于图形编辑、文本编辑，以及“专用”（special-purpose）编辑（比如为计算机电影动画编辑照明序列）。一个通用的编辑器可能需要进行太多的抽象。针对图像、文本和电影动画照明，编辑抽象中没有足够的共同性来从对三者的同时考虑中获益。

领域抽象的层次

在领域分析期间，我们进行全局性的抽象，并局部地进行具体化。“具体化”（reification）意味着：根据抽象的实例是如何用在现实世界中的来讨论抽象。对于系统族的每个成员，我们使用通用抽象来满足具体的需求。例如，如果我们正构建一个运行在 Motif（图形）下的文本编辑器，我们将此应用当成需要 X Windows 一样来对其进行讨论。需要运行 OpenLook 的其他应用也一样。**XWindows** 是超出了这个领域的抽象。在讨论每个领域时，我们认为它使用了被称为“**XWindows**”的东西，尽管 **XWindows** 可能只作为一个分析抽象存在。使用一个来自领域的抽象，就好像它是具体的构件块，这就是“具体化”。

领域分析的抽象应当比普通分析期间考虑的抽象更加抽象，但只在它们更通用的程度上。领域分析的目标是不仅要提高抽象的层次（作为认识问题的辅助工具），还要提高通用性和可应用性范围的层次。“通用”并不是“含糊不清”的意思，而是意味着我们抑制了架构上相关的细节，强调对领域很重要的共同性。如果一个设计表达的差异性比另一个设计更少，那么它就更通用。我们要捕获共同性和差异参数的本质。我们枚举出描述了应用领域特性的差异参数，而不是枚举出差异性。例如，我们不会说“我们需要 `Stack<int>`、`Stack<Number>`、`Stack<Message>`……”，而是会说“我们需要 `Stack<T>`，这里的 T 是某种类型”。这种用参数表示的（即“参数化”的）设计比枚举更加通用。但它并不含糊不清——我们可以为其写出每一位源代码。

为了提升抽象，我们要减少规格说明。也就是说，相对于更具体的类来说，更加通用的类拥有更少的成员函数，或者拥有更少的限制性接口。有时也可以通过这种方式使抽象更加通用。但我们也可以用单个通用模板（它捕获了族成员所共享的接口，其参数控制了各个例化）来替换一个相关的族、具体的类。模板可能会比原来的各个类抽象的层次低一些，但它肯定更加通用。作为一个例子，我们考虑用不同的加密算法（又一次针对不同的市场）来参数化文本编辑器的 File（文件）领域。

简而言之，我们要寻找系统的族。通过扩展到相关系统的族，我们减小了需求变化的风险。如果从一个更具体（非常具体）的架构“派生出”一个具体应用架构，则即使

是很小的需求变化都会“派生出”主要系统结构或接口中的变化。如果要让结构更加通用（定位于领域而不是某个具体的应用或客户），架构将更有可能随着系统进化的冲击一起发展。我们可以跨空间和时间“复用”设计——这是领域分析创始人[Neighbors1980]的主要意图。

4.1.3 应用领域分析和方案领域分析

在许多传统的设计方法中，应用分析意味着在开始设计和实现之前要做的事情。在多范型设计中，我们运用对实现领域的预见来获得益处。我们将这种方法看作是为对象范型所声明的优点之一的扩展（论据是假设编程语言能够表达“对象”）。在分析期间，我们寻找“对象”作为抽象的单位。在从分析到设计和实现时，我们会惊奇地发现：我们可以用实现语言来表达分析的抽象。

对象范型是最早普及这种方法的范型之一。经典的软件范型依赖于中间形式、状态图、数据流图以及在分析与设计之间转变后涉及转变的许多其他制品。对象范型试图用一种“以一当百”的方法来线性化应用领域和方案领域。

在多范型设计中我们同样这么做，只是我们拓宽了实现时设计者可利用的设计表达方式。我们打开了所有的语言特性，而不仅仅是面向对象的语言特性。

4.1.4 领域分析活动

领域分析包括三项主要的活动：识别业务领域、将业务领域分成多个子领域、并在每个子领域中建立抽象。多数软件开发努力都是由单个客户的需求所形成的（或者说由此驱动的）。在我们进行领域分析时，有一点很重要，那就是要将分析拓宽到多个客户的需求。这些多客户视点可帮助我们识别关键的业务领域。在多数业务应用中，有经验的实践者可以凭直觉识别领域。客户可以帮助我们精化所关注的领域；涉及到客户的领域才是好的领域。

子领域是对通用业务领域或企业的成功很关键的“具体的”业务或技术范围。识别子领域是多范型设计的最关键部分。不幸的是，设计的这一部分也是最难以规范化的（也就是说难以用统一的方法去解决）。将一个应用分成适当的子领域，这要依赖于所分析的业务和应用的相关知识。潜在的客户，甚至假想客户都可以调整领域共同性的列表，并向其中添加条目。在前面“编辑”的例子中，我们有一个带有子领域 **Database** 和 **Linear File** 的领域 **Persistent Storage**。每个子领域都可以用其余的编辑代码来等效。但支持数据库检索和更新的抽象和算法与支持普通文件的抽象和算法大不相同。它们是不同的关注区域，值得拥有相应知识的开发团队的关注，所以我们将它们看作是独立的子领域。

历史上成功的业务子领域就为该业务提供了一个通向新产品设计的良好开端。对于一个有经验的设计者来说，子领域的划分是从领域的经验中来，而不是从任何正式的或严格的方法中来。一旦子领域划分完成，设计团队接着就可以通过使用共同性和差异性分析，对每个子领域进行领域分析。这些活动指明了系统架构和实现的方向。

有些领域缺乏经验基础，不能依赖直觉作为划分的标准。那么如何为一个新的业务，或者为已存在的业务中的一个全新的问题找到它的领域？这个问题的解决方法是：在将领域划分成多个子领域以前，先在系统层次中进行一个非正式的共同性和差异性分析。共同性通常为领域划分指明了方向。

最好的子领域是互不相交的（disjoint）。“互不相交”非正式地暗含了这样一种意思：领域有明显的差异参数，每个领域都敏感于其自身的“固定”业务驱动。一个好的领域或子领域形成了相关抽象的族。将一个子领域当作一个架构和设计复用的单位是很有用的。只有当子领域的软件与其他软件完全解脱以后，它们才可能相互独立地进行复用。如果没有充分地分离开来，他们就无法与其他的软件进行整合。但为了复用和简洁起见，我们要尽量减少这种依赖性。第 7 章就聚焦在针对这种独立子领域的多范型设计上。尽管由于业务的原因，或者因为复用的好处仍然比耦合的价值大，我们必须保持领域之间的相互分离，但有时领域之间的高度耦合是无法避免的。在这些情况下，我们必须同时从多个领域中解析出有关的差异参数，并管理领域之间的依赖性。在“编辑”的例子中，**Text Buffers** 领域可能与 **Editing Language** 领域（可以处理编辑命令以及命令与屏幕更新之间的交互）交叠。这两个领域都是合理的关注区域，并依据领域自身形成了恰当的领域，但每个领域都必须放在另一个领域的语境中考虑。我们来考虑一个简单的“行编辑器”（line editor），其文本缓冲区只管理单独的行。“可视化编辑器”（visual editor）可以管理一屏或两屏大的缓冲区（或每次可以管理整个文件图像）。“流编辑器”（stream editor，比如 **sed**）需要一个文本缓冲区，该缓冲区确实是一个流缓冲区。

第 8 章描述了领域之间依赖性问题的解决方案。第 7 章和第 8 章的方法的成功依赖于被尽可能解耦的子领域（这也是我们从最早的分析阶段考虑得来的内容）。

4.2 领域分析中的子领域

前几节中我们注意到：传统开发是从探究单个客户的需求开始，期望获得能够解决分析中表面上的问题的单一集成的解决方案。下一步通常是将系统划分成多个独立的、可管理的部分。范型是帮助我们找到系统中这些部分（也称为“块”）的规则的集合。这些部分可以是自备（self-contained）的系统，该系统可以是能单独销售、或在给定的规范

中能单独管理的构件块。为了找到这些构件块，设计者可以使用对象范型、数据库建模、数据模块化、过程分解，或所选择的其他范型。传统设计通过使用一个主范型来分解一个单独的应用。在多范型设计中，我们按照领域工程的原则对此进行了扩展。但我们不会直接从单个范型跳到多种范型。首先，我们要回顾一下设计作用域的问题。

系统领域分析的作用域定义了一个语境，我们正是在这个语境中寻找抽象的。这个作用域比单个系统的要宽。我们可以再次应用一个主范型来从领域中获得抽象。但由于作用域中的领域很宽，所以单个范型可能不足以表达领域的抽象。（实际上，这对于使用了传统分析的复杂系统也一样，只是在领域分析中它变得更加明显）我们打算直接对领域使用多种范型。但问题是：我们从何处下手？

大部分大型的问题领域都是从多个小的子领域“编织”而成的。例如，金融贸易领域包括子领域：金融工具、人机接口、内部进程通信、数据库及其他。在电信领域中，我们找到了子领域：呼叫处理、恢复、管理和维护、编制账单及其他。在文本编辑器领域中，我们找到了子领域：缓冲区管理、人机接口、文本文件及潜在的其他子领域。这些子领域自身又是一个领域（“活动、形式或功能的范围”），此领域为设计者提供了一个焦点。

有些领域可以是模块或子系统。我们来考虑前面列出的一些领域。金融工具和数据库可以相互交叠，就像 **Text Buffers** 和 **Editing Language** 一样，电信中的大部分领域也是这样。后文中我们将会看到，甚至在简单的“文本编辑”的例子中领域也按不满足自然模块边界的方式相互交叠。

当高层次的领域分析牵涉到作为生产线的系统族时，我们可以跨应用领域来寻找子系统族。子系统可以不是单独的自给自足的产品，但它仍然可以为业务领域捕获设计和实现决定。Window（窗口）系统是一个很好的例子：它们很少独立存在，但它们却可以为一个定义明确的区域捕获设计决定[①]。在文本编辑器的领域分析中，“窗口”作为一个常见的抽象被发现。不同的文本编辑器的窗口特性可能不同，所以窗口包形成了一个族。这些族自身描述了一个领域的特性。我们称这些族为子领域，从而将它们区别于更宽的业务领域（这是当代领域分析的流行主题）。在多数语境中，文本编辑将是一个包含窗口、文件和命令处理子领域的领域，如图 4.1 所示。在类似文本处理这样的更宽的领域中，文本编辑器可以是一个子领域。我们要让文本编辑器的族支持给定的文本处理应用。子领域将是共同性和差异性分析中我们的主要焦点。

① 大部分的范型在捕获所有的设计决定，尤其是捕获设计基本原理上都很薄弱。这里，我们不是要介绍所有的设计捕获问题。此处，针对类 Window 的代码实际上已经可以为具体的应用和领域捕获设计目标，尽管此代码的用户或读者还不能对所有的设计目标进行反向工程（reverse-engineer）。

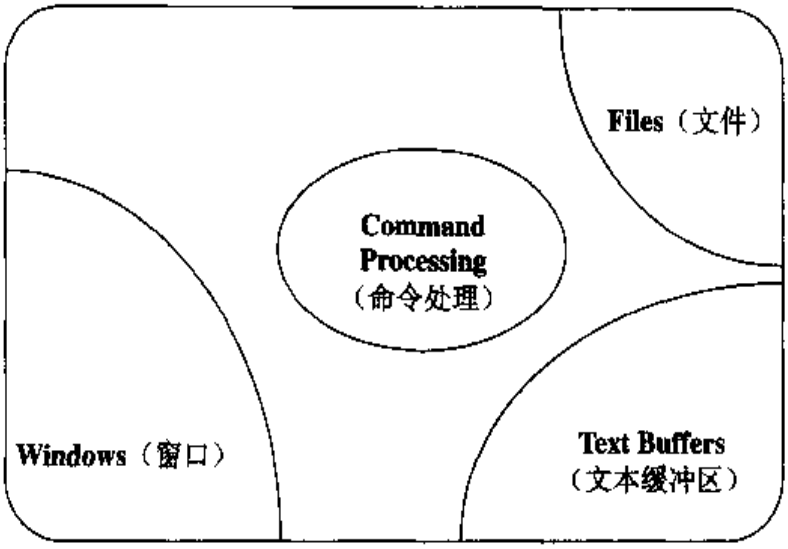


图 4.1 领域 **Text Editing** 中的子领域

子领域超越了应用。在特定实现中，我们为子领域捆绑差异参数以生成子系统。子领域是可以捕获经常跨业务领域出现的类似子系统的共同性和差异性的抽象。这些子系统形成了一个族。子领域分析描述了这个族的特性，捕获了其共同性和差异性的属性。我们可以适当选择差异参数来生成任何族成员（任何子系统）。框架结构通常是捕获跨族成员的共同性的适当方法，同时它提供了表达各个子系统的差异性的参数和接口（第 4.3.1 节）。在编辑器的例子中，领域 **File** 可能要依赖于控制缓冲方法、字符集和加密的差异参数。为这些参数选择适当的值就可以为具体的应用和市场生成具体的族成员。

4.2.1 领域分析和复用

正如前文所述，好的领域分析是复用的一个关键。因为传统分析聚焦在眼前的应用上，所以其抽象将基本不能服务于其他应用。真正的复用需要一个包括多个市场调度单位的更宽的视角。领域应当扩及业务的边界，但不要超出太远。如果领域太宽，它们就很难实现如果领域太窄，则它们的有用性就受到限制。

好的系统设计者通常都知道与他们所熟知的业务相关的子领域。也就是说，他们知道产品中接连重复出现的共同性的模式。这些直觉子领域形成了分析的第一个层次的“块”（bulk）。共同性分析帮助我们发现更细致的子领域，或者与新的应用区域有关的子领域。一个好的领域展示了词汇表、结构、名字、行为和用例的模式（第 2.3 节）。给定的领域可能没有展示出客户（例如，金融贸易人员和交互文本编辑工具的编写人员都使用人机接口）、制造商（有许多窗口厂商）或目标应用的共同性。

偶然选定的子领域不只适用于给定业务中许多应用，还可以扩展它们，使其可以适

用于企业中的多个业务。数据库、人机接口、内部进程通信和文件管理都是可广泛应用的领域的很好例子。这就意味着我们可以突破当前关注的客户领域，到达我们的业务中或有关业务中的其他领域，以形成一个更宽的抽象族。当然，许多子领域仍然是各个业务领域所特有的。例如，我们不能指望电信管理和维护的抽象中许多都可以广泛地应用于其他业务，也不能指望高度并行的图像描述的范型可以对经典数据分析程序贡献更多。

业务领域分析的输出包括下面这些：

- 分析所捕获的产品族的一种特性。
- 所关注的子领域的列表。

这些制品可以直接驱动方案领域的结构；当然，从应用领域分析的一开始就要注重这一点。

为一个广泛的领域封装了共同性并可以用非常特殊的差异性调整的软件制品为复用提供了坚实的基础。对象范型通过继承来支持这种复用。也就是说，新的派生类复用了基类的设计和代码。我们使用多范型设计而非类型特殊化来表达复用的形式。

甚至在划分子领域的时候，优秀的分析员就开始密切关注可用的实现技术。只有当确定了分析的方案结构时，分析员才能更充分地理解如何将领域划分为多个子领域。我们将在第4.4节详细讨论这个问题。

多范型设计只是软件复用的一个技术启用者。好的复用和好的领域分析要依赖于对业务需求的充分了解和洞悉。复用本身是一个带有业务目标（比如提高有效性和减少上市的时间）的业务实践。这些考虑超出了本书的范围，但书中将提到它们，以调和读者在纯设计角度所产生的期望。最重要的复用问题涉及到社会学、组织机构和市场的问题，这些也超出了本书的范围。还有一点也很重要，就是要认识到一个综合的复用程序不仅依赖于相关的技术，还依赖于相应的奖赏制度、组织问题和经济模型。欲了解这方面的详细内容，请参考Tracz[Tracz1995]的实践的和易于理解的著作。

4.2.2 子领域模块性

理想选择的子领域是模块化的子领域。也就是说，划分的结果是互不交叠的块。我们用子领域将大系统的复杂性分解为更小的、可管理的块。这些块相互独立和内聚，具有很强的可管理性。有些情况下，模块化可能比较困难，分解是否成功要取决于目标领域。实际上，领域越内聚，划分起来也就越困难。我们无法也不应当对应用领域的属性进行处理。我们只能精心地选择“刻刀”，以便从问题定义中获取最好的可能划分。有了这些子领域块，我们就可以开始对每个子领域块进行共同性分析了。

我们可以对业务领域系统的整个区域进行领域分析，但只有对子领域的领域分析才有意义，这有两个原因：第一，许多业务领域都太宽泛，无法进行共同性分析；第二，虽然领域内的抽象是紧密耦合的（比如定点设备的键盘和窗口），但跨领域的耦合限制了它们的抽象的复用能力，以及独立进化领域的能力（正如第 4.14 节所述）。

当代的多数分析方法也建议：设计的一开始，应当用一个高层次的划分将目标业务领域划分成多个可独立维护的块。简单的过程分解从过程层次开始入手，简单的面向对象的分析从对象开始入手。当代的多数设计方法都使系统进入了一个单一的抽象模式。其他的方法（比如 Booch 的方法）都从子领域开始入手[Booch1994]。更重要的一点是要找到将要带来耦合和内聚的最好模式的共同性和差异性的模式。这些将反过来提供良好的软件进化动力。

领域常可以使用常规的管理结构（比如子系统，参见[Booch1994]）来捕获。但领域是设计的逻辑单位，它并不是总能够映射到定义子系统的毗邻的代码体上。（第 4.3 节后有一个例子）。领域甚至可能相互交叠。多调度问题就是面向对象的程序设计中这样一个共同的例子。我们来考虑将 **Text Buffers** 领域所管理的内容输出到 **Persistent Storage** 领域所管理的一个文件的代码。哪一个领域“拥有”这段代码？代码结构和算法同时依赖于这两个领域。

还有一点需要重点注意：所有的子系统都是领域，子系统是代码的传统管理单位，而不是设计的逻辑单位。例如，企业可以选择某个领域来适应市场利率，甚至适应系统模块性和配置的市场份额，尽管此领域涉及到代码的多个部分。将领域放在第一位考虑将会使管理其进化变得更加简单。

4.2.3 迭代和层次

软件传统甚至是更深厚的西部文化的传统，通常会引导我们用自上而下的方式来解决问题。这种层次方法很适用于流行的业务结构（它们通常是分层的）当企业要在合理和可能的地方将软件结构和业务结构结合在一起时，这种方法是很好用的。实际上，驱动领域最初划分为多个子领域的很多直觉（如软件工程原则）都来自业务经验。

在大型系统中，子领域结构可以递归。从领域到子领域的初始划分可能遗留下来一些不可管理的大型块，需要在后续工作中“再分”（subdivide）。当抽象比市场的词汇表中的某个条目还详细时，就应当停止再分，此时开始选择适合将子领域分成可管理的设计块的范型。

严格分层的方法会留下严重的“盲点”，尤其是在新的领域中。在处理低层次的子领域时，设计者通常会获得对高层次领域的新认识。设计者应回到高层次的设计划分中，

利用这些新的认识。例如，在 **Database** 和 **Linear File** 领域中工作时，可能获得了关于差异参数和共同代码的认识，对 **Presistent Storage** 领域的分析可以用到这些认识。**Persistent Storage** 领域的客户软件的进化可能会暗示它们与 **Persistent Storage** 的合同中的变化，这种变化反过来会影响其子领域的结构。

一旦我们将应用领域划分成多个子领域，领域分析就不是此后的第一项活动。对解决方案的认识的不断加深逐渐形成了我们思考问题的方式。如果领域分析的结果是决定性的，这就存在危险性。任何成功的开发过程都必须包含迭代。

4.3 子领域的结构

我们必须将子领域分成若干可以单独开发的部分，这是范型所熟悉的领地（比如过程范型和对象范型）。我们如何来鉴别一个好的划分，一个分析问题的“最佳解决方案”究竟是什么意思？范型通常要应用可靠、实际的耦合和内聚原则。将模块内部的内聚增到最大是很有益处的，就像将模块间的耦合减到最小一样。范型指导我们创建满足这些指导原则的模块。我们可以使用 Parnas[Parnas1978]意义上的模块，或者是其他范型中的过程、对象或抽象这样的模块。

如果挖掘得再深一些，我们会发现：耦合和内聚的原则满足了更深层次的需求——所创建的软件容易编写，并易于理解和修改。单一的预想方案范型会妨碍我们对问题或领域自身中直接的结构的理解。

为简洁起见，我们要努力保持给定应用或应用领域内的子领域不相交。在划分子领域时，我们挑选最现代的分析方法来完成它们的第一道工序：将问题分解为抽象。多数范型寻求将一个系统缩减为多个部分，这些部分可以被单独地理解。传统方法在某一时刻对一个系统只应用单个范型，并可能选择能够给予我们最直观划分的范型。我们使用一种范型将系统分块成平衡结构、行为和绑定时间的多个单位。我们将这些部分当成空间分布的“物理”实体（跨源代码或贯穿一个正在运行的系统或程序的内存区域）。

C++的一个重要原则就是：没有一个单个范型能在所有的时候都工作得最好；这条原则也适用于应用领域和方案领域。如果有应用不同范型的机会，我们就可以在单个设计中使用多种范型。这些范型可能会在一个领域中相互交叠。在给定的领域中，相对于领域的各个“部分”来说，我们更关心在多个维度中贯穿系统的共同性的模式。如果要将系统分解成“物理”独立的模块，多数范型都会聚焦在系统的一个物理视图上。除了物理视图以外，我们还期望得到一个“逻辑”视图，让我们能清晰地看到贯穿任何单个物理划分的重要抽象。我们要寻找领域中有意义的逻辑分组，而不是寻求一种严格的划分。

这就为超越了对象（或过程、模块）的重要抽象留出了空间。

我们发现，普通C++程序设计中的逻辑划分和物理划分之间存在简单的冲突。例如，一个系统可能有一个将自定义类型的实例转化成字符串的函数族。共同的习惯是为每种类型指定一个asString函数。这些函数是逻辑相关的——它们都有相同的名字和系统的含义，但每个函数适用于不同的类型。从架构上来看，每个这样的函数都可能与其所适用的类型相关联。我们可以将它们写成friend（友元）函数：

```
class MyType1 {
friend string asString(const MyType1 &s) { . . . . }
    . . . .
};

class MyType2 {
friend string asString(const MyType2 &s) { . . . . }
    . . . .
};

. . . .
```

如果asString的每个实例都可以用类似的结构实现，那么我们就可以创建可根据其他成员函数来编写的一个单独的模板实例。代码的这种闭合拷贝“生成”了有关函数的族：

```
template<class T>
string asString<T>(const T& c) {
    // really dumb implementation of asString
    string retval;
    char *buf = new char[BUFSIZ];
    strstream str(buf, BUFSIZ);
    str << c;
    str >> retval;
    return retval;
}
```

如果考察我们想向其中添加asString行为的所有的类，并注意到拥有类似签名的这些类型，那么我们就会超越函数将共同性拓宽到这些类本身。我们将使用继承层次来表达此共同性：

```
class Base {
public:
    virtual string asString() const;
    . . . .
};
```

```
class MyType1: public Base {
public:
    string asString() const { . . . . }
    . . . .
};

class MyType2: public Base {
public:
    string asString() const { . . . . }
};
```

当然，如果前一个用例中的所有 asString 函数的结构都相同，那我们就可以用基类中的单个闭合拷贝作为它们的实现。所以，尽管 asString 是一个合法的逻辑抽象，但我们会发现软件结构中它的实现并没有模块化。

多范型设计承认存在单一的范型无法满足需要的情况，因而寻求将多种范型编织在一起。大多数时候，单一范型可以为某个应用表达出最佳的抽象。对象范型可以为许多应用提供最佳的匹配。但其他范型仍然有它们各自的地位，其中有些比对象范型更适合于某些应用。如果每个领域都可以用单一的范型来管理，那么我们就可以使用第 7 章介绍的简单的多范型方法。有时某个领域多种范型结合在一起工作得很好，能够产生所分析问题的最佳解决方案。对范型设计的这种更复杂的形式将在第 8 章中介绍。

为说明多重划分是如何一起工作的，我们来考虑在 Schwartzbach 和 Palsberg[Palsberg+1994]的设计范型中所提出的补充类型和类抽象。早期面向对象的设计将“类”和“类型”看作是相同的。Schwartzbach 和 Palsberg 提出了许多基础性的内容来帮助我们区分类型（行为）和类（实现）。这以后，设计任务就不仅要识别类型而且要识别类，以实现这二者以及二者之间的映射关系。Trygve Reenskaug 的基于角色的设计与这里的目的类似[Reenskaug1996]。在多范型设计中，我们努力地将设计视野扩展得更远。

4.3.1 以框架作为子领域的实现

“框架”是一个共同性不断增加的软件包，它捕获了源代码和目标代码中的软件架构。它是一个被部分填满的实现，还存在一些“洞”，需要基于每个应用来填充。因此，框架是一个抽象，它描述了有关实现的族的特性。

框架描述了问题的解决方案，所以在分析期间就聚焦在框架上为时过早。本书在此（“分析”章节）讲述它们是为了让读者对设计的可用终端产品掌握得更牢固。

框架通常捕获了整个领域所共同的实现。最共同的框架可以支持用户接口。Trygve Reenskaug 的“模型－视图－控制器”（Model-View-Controller）是一个最通用的人机交

互模型，它是相关框架族的基础。软件公司的专有框架支持了与业务方向密切相关的软件族。

典型的框架定义了一个接口，用户为其提供了一个实现、抽象基类（用户从中派生出实现类）或框架中的类之间的耦合关系。如果一个基类通过规定用户必须定义的函数的接口来指引实现者，那么这个函数就是一个差异参数。如果类的派生被用来捕获特定族成员的实现，那么基类和派生类之间的结构和行为上的不同就代表了差异参数。类（或它们的对象）之间的登记模式捆绑了对实函数（为具体的应用而编写）的职责分配的抽象概念，与 Smalltalk“模型－视图－控制器”框架中的类似。在每个这样的实例中，我们用用户提供的结构来“参数化”普通的框架，以生成一个族成员。

有时，单独的子领域能够很接近地映射到框架上。好的框架可以独立地交付、维护和进化。第 7 章中我们所讨论的子领域就可以很接近地映射到框架上。紧密耦合的子领域很难作为独立的框架来进行管理。一个丰富的框架通常合并了多个子领域，有时通过使用多种范型，来处理宽作用域的业务问题。框架常常来自于范型之间的充分交互，这将在第 8 章中讲述。

4.3.2 子领域的分析活动

第 4.1.4 小节为将一个业务领域分成多个子领域建立了一些原则。这些子领域中的每一个都必须对在一个 C++实现中能够表达的共同性和差异性的“压点”进行分析。这是多范型设计的一个主要焦点，正是此活动将多范型设计方法与其他方法区分开来。

为做到这一点，我们应用第 2 章和第 3 章中介绍的方法。第一步是识别领域的共同性和差异性。尽管这些都是在第 2 章和第 3 章中分开介绍的，但在实践中要一起来完成。共同性帮助揭示了差异性，反之亦然。

内省和清楚的定义是好的领域分析的关键（第 2.5 节），我们必须问自己许多关于共同性分析的问题（第 2.2.2 小节）：定义是否清楚、完整和一致？共同性是否完整、正确、精确和一致？差异性角度也同样重要（第 3.9 节）。差异性是否延伸到了领域的边缘？今天我们定义差异性时和团队预期时它将如何变化？差异性是否完整、简洁和一致？对差异性的其他审查可以通过差异参数进行（第 3.2 节）。它们是否完整、是否与差异性一致、它们的默认参数是否适当、是否在领域范围之内？

并且，我们不能丧失全局观点：子领域产品能否作为产品上市？答案不会总是肯定的，市场也不可能总是广阔的。但子领域应当内聚，并应当提供可以帮助业务的直觉功能性。

4.4 分析：全景图

在本章中，我们从激发需求以将分析扩展到领域分析开始。我们通过寻找超越我们的业务的产品的软件族或我们期望超过给足产品版本的软件族来实现这一点。

理论上，我们可以将领域分析描述为介于两个极端之间的方法的区域。在第一种极端的方法中，我们将问题领域分析与方案领域分析分开。我们可以在不管如何表达解决方案的情况下来分析问题领域。这样做有一个优点：我们可以推迟实现的细节。但这是有风险的，所交付的分析在合理的开发技术和文化的约束内可能无法实现。

在第二种极端方法中，我们从一开始就考虑了实现技术。例如，我们将依据 C++语言结构塑造出所有的领域抽象：类、模板和函数等。

这两种极端的方法都无法提供令人满意的工作点，但它们对于协调分析方法和随后的实现策略依然很重要。一个好的分析员将会从实现技术的某些概念出发，但又不让这些知识支配所创建的抽象。当多范型设计方法用在子领域中将分析结构和编程语言的结构结合在一起时，映射就偶尔会不方便甚至难以处理。当难以使用 C++结构表达一个子领域时，就需要重新考虑领域划分。在极端情况下，重新考虑是否使用 C++作为实现技术将很有益处，尤其如果其他的一些实现技术在过去已经证明了自己的实力。例如，基于规则的系统应当考虑使用支持专家系统的工具（比如 Prolog），而处理大型数据集合的管理和当前访问权限的系统时应当使用数据库工具。

图 4.2 总结了多范型设计的抽象活动。在本章中，我们已经聚焦在图的左上方的圆框上。我们从识别目标应用领域或目标业务开始，我们希望工作在其中。我们更多地依赖于我们的直觉（相对于任何规范化的或正式的方法而言）将问题分成多个子领域。接着，我们对单个子领域进行共同性和差异性分析，正如下图演示的对 **Custom Protocols** 进行的处理。这是一个迭代的过程，对共同性和差异性的处理并行进行，并收敛于子领域的第一次选择。

共同性和差异性分析一旦完成，设计者就必须将每个子领域结构与方案空间中的可用结构结合起来。要实现这一点，设计者必须理解实现空间的结构。实现语言自身是一个领域。设计和实现都是将应用空间与方案空间的共同性和差异性结合起来的艺术。图 4.2 右上方的圆框代表方案领域的分析。我们将在第 6 章深入讨论此圆框中的内容。将应用领域和方案领域的结构结合起来就获得了以下方圆框所示的框架。第 7 章和第 8 章将讨论结合这些分析的方法。

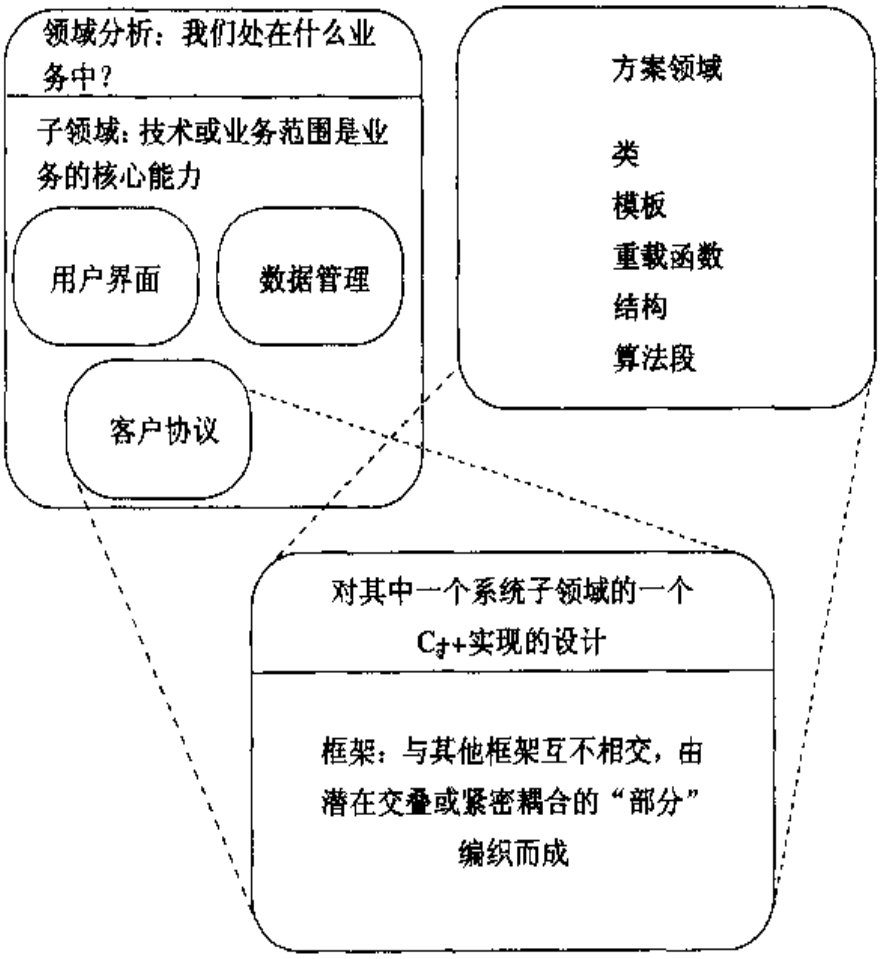


图 4.2 分析的领域

当我们发现所选择的实现技术并不适合于应用领域的共同性和差异性分析时，我们将要重新回到应用领域的分析或（和）方案技术的选择上来。多范型设计（与所有的好的设计一样）从根本上是迭代性的。找到能够雕刻出各方面都适当的解决方案的理想“刻刀”是要花时间的。相对于单一应用的简单分析，好的领域分析就更耗时。消耗这些额外时间的好处就是：产品可能具有更广泛的适用性，并可以更好地经受时间的考验。

即使如此，这个图还是不太完整。最终的产品是一个框架结构，可针对具体的应用对此框架结构的差异参数进行绑定。最后的这一步是针对每个应用而进行的，给定的应用领域中的每个子领域都需要进行一次此步骤。随着时间的推进，领域经验将反馈到领域分析过程的更深层次的迭代中。

4.5 小结

我们努力进行领域分析，而不仅仅是分析，以提高系统能够很好地进化的可能性。

一旦确立了领域的宽度，我们就努力地将其分解成多个子领域。分析子领域可以使用共同性和差异性分析。有时共同性和差异性分析将系统清晰地显示为一个整体。此时，子领域划分就可能需要重新地审视。当共同性和差异性分析收敛于一个领域时，实现抽象就开始显现出来。

到这里，我们可以开始将共同性和领域分析的结构结合到 C++所支持的结构上去。但首先，我们要看一看作为共同性和差异性分析的具体实例的面向对象的分析，并将其作为一个扩展示例研究一番。

第5章
面向对象的分析

在本章中，我们将对象范型当作问题领域分析方法的一个例子。我们根据前几章中介绍的共同性和差异性分析的原则进一步展开面向对象的分析。我们假定对象范型是最合适的范型，通过使用共同性和差异性原则来分析一个应用，从而得出设计抽象。

5.1 关于范型和对象

Kuhn 在其代表性著作中首次推广使用了术语“范型”（paradigm），随着对象范型的普及，此术语成为软件中的一个常用词。范型是用来理解周围的世界的惯例集合。在软件中，范型形成了我们明确表达抽象的方法。世界是由什么组成的？世界可否划分为过程、数据记录、模块、类和对象？

在实践中，一种范型对规则、工具和惯例进行编码，从而将领域划分成多个可以被单独理解的块。范型的目标通常包括笼统地称为“模块性”（modularity）的指标，这就意味着分析应当产生内聚且去耦的抽象。如果抽象是相互独立的，这样它们之间就不会相互影响，它们的所有者就可以独立地演进它们。如果假设我们可以通过演进系统的某些部分来演进整个系统，那么范型就可以通过创建“变化的群岛”来帮助我们。

5.1.1 类和对象

类和对象是对象范型的主要抽象。内聚和耦合是基本的划分标准。二者是从模块范型和结构化设计[Stevens+1974]中借用而来的。我们根据有关职责的分组（有时根据一个内聚的数据结构）来形成“对象”。“类”是一种编程语言结构，它描述了具有相同职责

的所有对象，用同样的方式实现这些职责，并共享相同的数据结构。“类”也用于设计抽象，作为“抽象数据类型”（Abstract Data Type）非正式的同义词，尽管本书试图忠于这些术语的更精确的定义。使用了类和对象的程序通常称为使用了“基于对象的程序设计”。

5.1.2 Liskov 可替代性原则

相对于面向对象的程序设计而言，对象范型更进了一步。它不仅为将领域分解为多个部件提供了原则，还为将这些部件分组为更高层次的抽象提供了附加的原则。这些原则与“子类型的导出”（subtyping）和“继承”（inheritance）有关。术语“子类型的导出”常用于描述分析和设计抽象之间的关系，而继承则是用于反映支持面向对象程序设计的大部分语言中的导出子类型的一种机制。（有些语言，比如 self[Ungar+1987]使用“委托”（delegation），而不是“继承”。）我们使用严格的行为一致规则将类分组到继承层次中。也就是说，当且仅当类 B 的对象可以替换依据类 A 所编写的程序中类 A 的每一个实例，并且没有改变原程序的行为时，类 B 才可以从类 A 派生而来[Liskov1988]。这就要求我们合理地设计我们的类，并且编程语言要通过生成替换派生类对象的“工作”来配合。允许程序将不同类的对象当作同种类型看待的语言特性是“多态性”的形式之一。

5.1.3 虚函数

在 C++中，继承和虚函数实现了多态性的这种形式。多态性的字面意思就是“很多种形式”。C++中的每一个类都代表了一种通用类型的形式或实现。例如，下面的声明描述了一个抽象数据类型接口（“抽象”意味着 operator+= 没有实现）：

```
class Complex {
public:
friend Complex operator+(const Complex&, const Complex&);
    virtual Complex &operator+=(const Complex&) = 0;
    . . . .
};
```

该类型可能采用以下几种形式之一：

```
class PolarComplex: public Complex {
public:
    Complex &operator+=(const Complex &other) {
        // . . . . complex polar stuff
    }
    . . . .
private:
    double radius, theta;
```

```
};

class CartesianComplex: public Complex {
public:
    Complex &operator+=(const Complex &other) {
        rpart += other.rpart;
        ipart += other.ipart;
        return *this;
    }
    . . . .
private:
    double rpart, ipart;
};
```

面向对象的程序设计的本质就是：依据对象在运行时间的形式选择一种类型的操作的实现。虚函数依据对象运行时的类型来分配成员函数的调用（比如Complex::operator+=）。

这是对象范型的一个相当正式的定义。更正式地讲，我们要设计松散地耦合但内聚的类，并将它们安排到规格说明的层次中去，将通用的类放在根部，较具体的类放在枝叶上。从编程语言的角度来看，面向对象的程序设计是："模块化编程"（封装）＋"例化"（创建给定的任意模块的多个实例的能力）＋"继承"＋"多态性"。Peter Wegner将支持对象范型的运行时类型的可替换性定名为"夹杂多态性"（inclusion polymorphism）[Wegner1987]，以便将其同多态性的其他形式区分开来，比如参数化类型和重载（参见图6.8）。

面向对象的分析从应用领域结构中派生出对象和类结构，同时将重点放在类层次[①]上。与任何分析活动一样，面向对象的分析依赖于分析员的直觉，他们知道以前什么样的对象和类是合适的，所以知道现在该怎样得出适当的对象和类。就笔者的经验来看，最有效的面向对象的分析方法都是非正式的。

5.1.4 面向对象的另一种定义：object orientation

我们可以依据共同性和差异性定义对象范型。假设从一个领域词典开始，面向对象的分析是建立在共同性分析的基础之上，我们在共同性分析中寻找类型的共同性。回想到类型是一个签名集合，签名在此集合中捕获了名字和行为（参见第2.3.2小节）。面向对象的分析抽象的第一步是建立抽象（按照行为分组）的族。所以，对于现代面向对象

① 有些面向对象的方法将继承推迟到分析阶段以外，我们称之为实现活动。有些继承分组不可避免地发生在分析期间，尽管设计的分析和实现考虑事项可以分来。

的方法的学习者来说，族的思想应当是比较直观的。

积极与消极差异性

我们依靠类型共同性的背景，在行为中寻找差异性。此差异性包括签名中的积极差异性。如果有些族成员显示了一些与族整体非共同的行为，这就是积极差异性，这些族成员将变成派生类。消极差异性没有包含在多数面向对象的设计方法和编程语言所支持的对象模型中。类型族中的消极差异性指一个将使用带相消的继承的实现，好的设计者会避免使用这种实现。对亲自体验过面向对象的方法的设计者来说，这是很明显的，但原因与第3.3节中所介绍的差异性有所不同。

寻找签名相同但实现不同的类型更常用。如果 Xwindow 和 MSWindow 都被适当地抽象到名为 **Window** 的领域抽象的层次上，则它们的签名是相同的。这两个抽象的行为明显不同，它们的实现也将不同。这是依靠共同的“语义”背景获得的行为中的差异性。

绑定时间

正如第3.5节中所述，绑定时间是领域分析的一个关键属性。面向对象的程序设计所关注的差异性是在运行时间绑定的。推迟绑定时间是平衡语义共同性和行为差异性的关键，也给予了对象范型抽象的能力。

默认值

默认值是领域分析的最后一个关键属性（第3.6节）。在任何的类型族中，较抽象的族成员为较具体的族成员提供了默认值。这些是继承属性的基本语义。继承机制通过一个定义了族成员之间的子类型的导出关系图来传递默认值。除默认值以外，子类型导出关系图（它几乎总是与 C++中的继承图相同）还捕获其他一些关系。子类型导出关系图的主要用途是捕获行为的特殊化，这可以根据有关类型的各个签名元素前置和后置条件的强度进行解释[Meyers1992]。多数情况下，（默认）行为的继承与子类型导出关系图是结合在一起的。

关于继承和子类型导出的文献有很多。多数情况下，二者是统一的。了解异常情况并据此进行设计是很重要的。异常作为一个族成员的集合出现，这些族成员或具有共同的结构但没有共同的行为，或具有共同的行为但没有共同的结构。这是特殊种类的消极差异性，我们将在第6.11节中对其进行详细讨论。最一般的方法是将实现和接口分成单独的层次。指针将实例跨层次地连接在一起。这是个经常出现的问题，所以其解决方案可以被规范化处理，此解决方案已作为“设计模式”（design pattern）被捕获。本书第9章中会更详细地介绍设计模式。

新经典的面向对象的方法（试图将传统结构化的方法中面向对象的语义融合在一起的方法）聚焦在结构的共同性上，而不是聚焦在行为的共同性上。它们将结构当作一个

分析考虑，而不是一个实现细节。基于结构的分析不太可能捕获架构和基于类型的方法（参见 2.3.1 小节）。尽管结构共同性和差异性常常是遵循类型共同性和差异性的，但仍然存在显著的例外情况。第 6.7 节的讨论中就有一个显著的反例。

简言之，对象范型是：

- 捕获称为类的显式编程结构中结构和行为差异的一种方式。
- 通过行为的共同性将这些类分组到抽象中的一种方式。
- 共享相似的类之间的默认行为的一种方式。
- 将运行时间的不同行为统一到可在编译时间知道、并与共同性分析的属性有关的接口中的一种方式。

当然，我们并不需要按照共同性和差异性分析来考虑对象范型。实际上，多范型设计根本不会因其自身的原因，或者因其自身的指标而考虑对象范型。对象范型只是用在多范型设计中，建立在共同性和差异性分析的基础上，当应用于适合面向对象的方案的问题时得出抽象（该抽象与通过当代面向对象的分析方法所获得的抽象类似）的一种方法。

使用经过检验的方法

如果多范型设计表明对象是主要的范型，好的设计者将寻求使用一种或多种面向对象的设计方法。面向对象的方法帮助我们处理多范型设计没有触及的重要的设计考虑，比如被继承的接口之间正确的子类型导出关系，以及对接口“胖”和“瘦”的充分关注。

5.1.5 面向对象的设计的适用性

我们怎么知道要使用对象范型而不是其他的范型？或者换一种问法：我们怎么知道在进行分析时要聚焦在面向对象的抽象上，而不是幼稚地去试探 C++所支持的每一种范型？如果应用的“压力点”与对象范型所支持的共同性和差异性的维度结合起来，对象范型会生成一个好的架构。（Pree[Pree1995]称之为“热点”，hot spot。）对象范型假定了类型和结构中的共同性，表达实现中的差异性。如果某个族中有多个成员可以映射到框架中，则对象范型就是合适的。

语言和范型

关于 C++与 Smalltalk 以及它们彼此对对象范型的支持程度已经有过很多的讨论。据称，纯粹的面向对象语言的优点之一是：它们将所有的设计抽象强制性地放在“塑造对象”的模子中。在纯粹的面向对象的领域中，其他范型居于二流地位，并且难以表达（除面向对象的术语以外）。

纯粹的面向对象的设计几乎没有，因为它们还要表达其他重要的共同性和差异性的

维度。C++很好地捕获了非面向对象的设计结构中的一部分。多数C++程序员自觉地使用一定程度的过程分解。FSM（有限状态机）是很多面向对象的设计中的一个部分，在C++中可以通过多种方式方便地实现。模板、重载和其他语言特性并不是本征地面向对象，也存在比C++所能表达的内容更多的其他设计结构。行业经验表明：对象并不能很好地表达“工作单位”（Units of work）。为什么？因为共同性和差异性及时处理相互关联的步骤和算法（连续调用），而不处理在意义上相关的结构和行为集。在电信设计中，重要的业务抽象（比如“呼叫”）或电信特性（比如“转送”和“呼叫等待”）很少能形成一个好的抽象，因为它们是“工作的单位”。当然，还存在一些数据结构、并行编程结构和其他重要的方案领域结构，如果它们根本就不是面向对象的，那么C++可以只按常规容纳它们。

多范型设计之所以重要，是因为它超出了对象来表达在C++中获得丰富表达的其他设计结构。要理解这句话的意思，也就是要理解将在第6章中讲述的解决方案技术的丰富表达方法。这也意味着要理解如何将应用空间同方案空间结合起来，我们将在第7章中讨论这一问题。前面提到过，有了范型的帮助，我们就可以通过演进系统的若干“部分”来演进整个系统，但这并不意味着理解了这些“部分”就等于理解了整个系统。对象范型给了我们对象、“部分”，而其他范型帮助我们理解形成系统的其他重要关系。C++还可以直接表达一些这样的关系。

究竟对象还可以用在哪些地方？对象范型的优点在于它捕捉了稳定的领域资源和制品。Windows是一个很好的例子。就电信领域而言，线路、干线和端口可以形成很好的对象。其他几乎每个领域中都有无数的例子。尽管如此，对象却不是普遍适用的。

在带有直接操纵接口的领域中，对象看起来工作得也很好。如果我们按照我们所组装和操纵的“部分”来考虑一个领域，对象常常是很合适的。图形图像库是耗时的典型例子。计算机辅助设计（CAD）程序是另一个很适合于对象范型的重要软件族。

为说明多范型设计和面向对象的设计之间的关系，接下来的几节将使用多范型设计来处理一个“面向对象的问题”。数字硬件设计是一个富含资源和制品的应用领域。为数字设计所编写的软件通常会使用一个直接的操纵隐喻（metaphor），它是对象的另一个“先驱”。

5.2 面向对象的共同性分析

通过使用共同性和差异性分析研究系统的对象，可以说明对象范型是如何适用于多范型并发的。我们用数字电路CAD程序作为例子来考虑。我们不是要设计一个电路编辑

器、仿真器或绕接发生器，而是要对整个电路设计领域进行分析。我们做领域分析时，数字逻辑设计是业务领域（参见第 4.1.4 小节）。

领域分析的第一步是建立领域词典，如图 5.1 所示。这一步不仅要建立一个帮助设计团队的成员进行交流的词典，还要建立多范型设计的基础构件块。

设计元素（Design element）：电路设计的任何电子或物理组件

网络元素（Network element）：带有一个或多个相关信号值的设计元素

门电路（Gate）：基本组合逻辑（非、与、与非、或、或非，等等）设计的网络元素

触发器（Flip-flop）：带有一位存储器的逻辑元素族

网路（Net）：两个或多个网络元素之间的电子无源连接

块（Block）：有关网络元素的逻辑分组，逻辑元素本身表现为一个网络元素。

标签（Label）：……

输入引脚（Input pin）：……

输出引脚（Output pin）：……

指令计数器（IC）：……

电路板（Board）：……

计数器（Counter）：……

寄存器（Register）：……

施密特触发器（Schmidt trigger）：……

线路驱动器（Line driver）：……

线路缓冲器（Line Buffer）：……

图 5.1　数字电路设计的领域词典

领域分析的下一步是找到子领域。这一步要依赖于直觉、依赖于我们对应用领域的了解。利用领域词典，有经验的设计者可以得出下面这些领域：

- 逻辑组件（**Logical Components**）
- 网路（**Net**）
- 端口（**Port**）
- 物理组件（**Physical Component**）

这些领域都是带有多个成员的族。每个族都通过主要的共同特性组合各个成员（族成员共享相同的共同性和差异性）。例如，“逻辑组件”是一个可以组合到设计中的所有逻辑元素的领域。设计者将“逻辑组件”当作设计的逻辑构件块使用。它们都有一个图示法，都有可以连接到网路的输入和输出信号，并且都可以查询实现它们的逻辑组件。它们的“行为”（behave）都是相同的。门电路、寄存器、计数器和触发器都属于这个领域。网路也是一个领域，其元素可以连接到逻辑组件或端口上。它们可以被改造、命名、创建、删除等。网路有多种类型：电网路、地网路、信号网路及其他网路。但在某些层次上，它们又都有相同的行为。其他领域如“端口”（与硬件构件块的接口）、“物理组件”（实际“芯片”，逻辑组件被打包到这个“芯片”中）的行为也类似，所以也对它们进行了分组。

我们本可以通过使用一个过程来优化每个领域中的共同性，并将跨领域的共同性减到最少，从而正式地形成这些领域。一个直觉性的划分已经足以包括正式的方法所可能缺少的见识。使用第 2.5 节的问题来审查共同性分析的结果仍然不失为一个好主意。

5.2.1 共同性分析

我们将“逻辑组件”分组在一起是因为它们具有相同的行为。我们对“网路”、“物理组件”和其他领域也进行了这样的处理。这些领域中所有的族成员也很可能共享相同的数据结构和实现。这些共同性是一个特性集合，这些特性使领域的成员形成一个族。

5.2.2 差异性分析

每个逻辑组件都会实现某些逻辑功能，但不同的逻辑组件所实现的逻辑功能是“不同的”。每个逻辑组件都有引脚，但不同组件的引脚数量各不相同。尽管组件都展示了结构和算法中的共同性，但它们也显示了结构和算法中的差异性。每个成员都有这些差异性，这个事实本身就是一种共同性，但我们使用差异性来区分各个族成员。

这些族都展示了行为和结构中的共同性，以及算法和结构中的差异性。这些共同性和差异性的模式定义了我们的分组标准，我们要用这些标准来进行抽象。这些标准形成了一个范型。那么，我们使用什么范型（不管是否是下意识的）将族成员分组到领域中？对象范型捕获了行为和结构中的共同性，以及算法和结构中的差异性。在这个例子中，共同性和差异性分析是断定我们应使用对象范型、每个领域应当作为一个继承层次结构的间接方法。

Booch 的“类类别”[Booch1994]与这里使用的“领域”密切对应。类类别包括一个类层次中的类，以及紧密相关的支持类和函数。我们可以使用一个类层次来组织大部分的领域抽象。例如，“逻辑组件”（**Logical Component**）可以用于组织“逻辑门电路”（**Logic Gates**）和“中等规模集成”（medium-scale integration，MSI）组件（参见图 5.2）。通常，这包括设计者用来实现一个硬件系统的逻辑功能性的所有组件。逻辑门电路捕获了与非门、与门、或门、非门以及其他很多门的共同性。“MSI 组件”包括触发器、移位寄存器、寄存器缓存、缓冲器、线路驱动器和其他“分块”的设计单位。

多范型设计并不处理继承层次的细节：基类和派生类是什么？哪些是公有继承哪些是私有继承？本书后面的“参考书目”中所列出的那些流行的面向对象的设计方法也可以很好地支持这些设计决定。切不可忽视了它们！

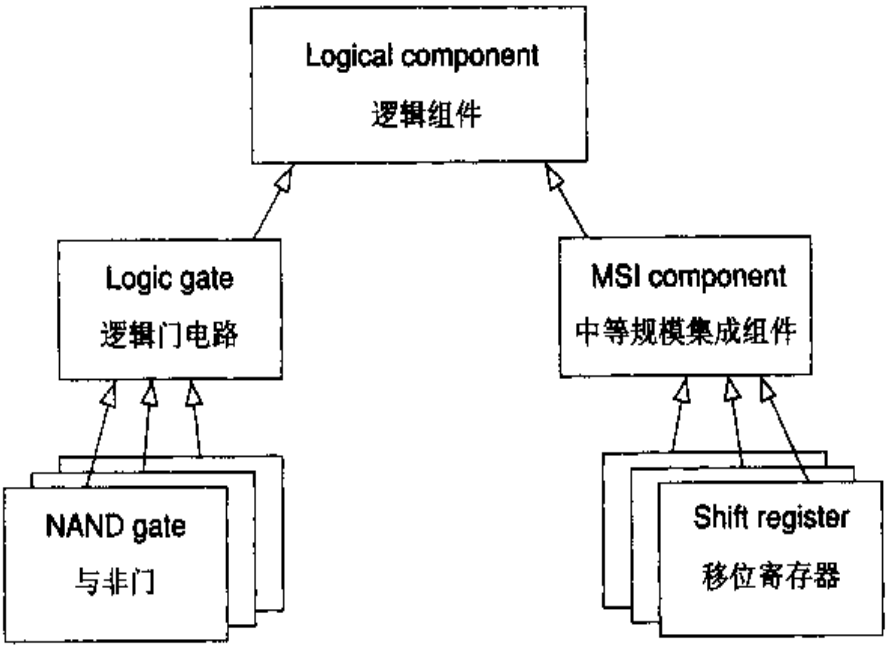


图 5.2 “逻辑组件”（Logical Component）领域

5.3 小结

本节介绍了简单的共同性和差异性分析是如何帮助设计者为设计选择适当的对象范型的。本章的目标是从面向对象的设计这个大家熟悉的角度来说明多范型设计的主要原则——共同性和差异性。

为电路设计和仿真设计设计一个包的问题远比本章所处理的要复杂得多。需要特别指出，我们忽略了很多重要的领域，比如图像、仿真、与模拟电路设计的接口和校验领域，以及很多其他的领域。甚至本章中所处理的领域也以间接的方式相互依赖，例如，门电路的逻辑和物理划分实际上不能相互分离。

下一章中将介绍一些基础知识，帮助我们跳出对象范型进入其他的架构风格。我们将从C++语言自身来逐步展开这些基本知识：C++语言支持哪些范型？

第 6 章
方案领域分析

本章我们将重新回到领域分析上，但这次讲述的是方案领域分析，而不是应用领域分析。应用领域分析的很多相同原则也同样适用于方案结构。这里我们所研究的特定方案领域是 C++编程语言的方案领域。

6.1 “其他”领域

在第 4 章中，我们研究了如何捕获共同性和差异性中的问题领域的结构。很多设计方法都只分析问题领域——然后就只涉及一种范型。尽管我们已经讨论过为什么使用多范型很重要，但还没有将此问题带入到方案领域。

为什么要研究方案领域？如果将设计看作是从一个问题结构导出方案结构的活动，那么仅仅理解问题是不够的——我们还必须理解方案领域。方案领域包括计算机架构、网络结构、人机接口和其他的子领域。但很多的方案领域的抽象都在编程语言中。正是编程语言的结构（或者更精确来讲，是编程语言表达共同性和差异性的方式）决定了方案领域的“形状”。本章将对 C++编程语言进行分析，以揭示其重要的共同性和差异性维度。也就是说，我们要从 C++的角度来分析方案领域。

有些读者可能对如此全面地介绍这些显而易见的语言特性有所不解，所以建议这些读者将本章当作对 C++所支持的设计特性的全面描述，而 C++可以帮助我们形成多范型设计下的设计和实现的词汇表。我们将在第 7 章中研究如何在这个设计模型雏形的基础上进行扩建。

6.1.1 分析和语言

不同的方案领域（比如不同的编程语言）需要完全不同的设计，即使这些方案领域具有相同的问题领域分析。我们来考虑在C++和Smalltalk中实现的一个简单的类Stack（图6.1）。这两种语言使用了不同种类的共同性和差异性，它们暗示着（甚至是可以说是导向了）不同的设计。C++设计的代码拥有一个形参T，这是Smalltalk设计中所没有的。在Smalltalk中，目标代码绑定到设计的时间比在C++中要早得多。

```
template <class T> class Stack:
     public ArrayedCollection<T>
{
public:
     Stack();
     ~Stack();
     T pop();
     void push(const T&);
private:
     int collectionSize, limit;
     T getElement(int);
     T *collection;
};

T Stack<T>::getElement(int index)
{
     return collection[index];
}

T Stack<T>::pop() {
     return getElement(limit--);
}
```

```
ArrayedCollection subclass:
#Stack
     instanceVariableNames:
          'collection
          limit
          collectionSize'
     classVariablenames: ''

Stack>>getElement: anIndex
     ^collection at: anIndex.

Stack>>pop
     |object|
     object :=
          self getElement: 1.
     collection replaceFrom: 1
          to: limit - 1
          with: collection
          startingAt: 2.
     collection at: limit
          put:
          self defaultElement.
     limit := limit - 1.
     ^object.
```

图6.1 对类Stack的一个C++设计和一个Smalltalk设计的比较

在分析过方案领域之后，本章将准备讨论如何将问题领域的结构塑造成方案领域的结构。这一步进行过后，多范型设计的基础也就建立完成了。

6.2 C++方案领域：概览

进行C++编程语言的领域分析意味着什么？我们要寻找C++编程语言能够描述的抽象族。现在我们有了进行族分析的工具，就可以直接描述下面这些族：

- 数据，对有关值的族进行分组的一种机制。

- 重载，对有关函数的族进行分组的一种机制。
- 模板，它对有关算法或总数据结构进行分组。
- 继承，它对行为相同的类进行分组（通常是公有继承，我们将在第 6.11 节中单独介绍私有继承）。
- 带虚函数的继承，它对行为相同但绑定比单独的继承迟的类进行分组。
- 预处理器结构，比如#ifdef，常用于代码和数据中的细微差异。

这些语言特性中每种特性都描述了一种范型的特性，描述了组织领域的一种方式。这些语言特性将代码塑造形成一个 C++程序，也形成了我们考虑和表达银行系统、电信系统、GUI 及其他应用领域的方法。下面几节将依次聚焦在这几个特性上，探究每种特性的共同性和差异性属性。最后在第 8 章中我们将应用领域和方案领域结合起来。

6.3 数据

C++提供了多种丰富的结构（从“内建”类型到“结构体”，所有获得完整的类对象的方式）来存储和组织数据。一个数据存储表示一个值族。数据的共同性和差异性分析很简单，在这里进行介绍的价值不大，但考虑到完整性还是(简略)介绍一下。

共同性：结构和行为。C++中的数据项有一个固定的结构，在源码编写时间建立，并在编译时间绑定到目标代码中。类型系统在编码时间和运行时间将有效的操作与数据值关联起来。

差异性：状态。各个数据项只是状态不同而已。

绑定：运行时间。在运行时间，数据项的值可以通过该数据项的任意非常量标识符获得改变。

例子：不同的颜色校正算法拥有不同的公式，公式的系数对应于不同的显示器制造商。这些系数可以当成类似显示器族的共同性，也可以当成区分显示器集的差异参数。

6.4 重载

经典的 C 没有提供对相关的函数进行分组的语言特性。C++提供了几个不同粒度的词汇和语义的分组：类作用域、名字空间和重载。重载支持一个作用域内的函数族。

共同性：名字、返回类型和语义。重载的函数形成了一个族，该族的成员通过名字和意义相关联。所有重载的函数都有相同的名字和一致的返回类型。C++没有强迫重载的函数之间语义的一致性，但一般都遵守这一准则来使用。

差异性：算法和接口。每个重载的函数都可以有自己的正式的参数、算法和局部数据，并都可以调用所选择的其他函数。函数是基于它们的接口而区分的（由所使用的编译器来选定）。

绑定：编译时间。在编译时间，根据语境选择相应的函数。

例子：设置窗口颜色的函数有多个。其中，有些函数可能带有来自 X 系统的 Xgcv 值。有的函数可能带“红－绿－蓝”强度参数。还有一些函数可能带有适合于其他窗口技术的其他种类的参数。

Stroustrup 的早期著作[Stroustrup 1986]中有一个典型的例子，即下面的代数乘方函数的集合：

```
int pow(int, int);
double pow(double, double);     // from math.h
complex pow(double, complex);   // from complex.h
complex pow(complex, int);
complex pow(complex, double);
complex pow(complex, complex);
```

6.5 类模板

C++模板是早期 C++程序员为捕获源代码的共同块（可以用宏参数来参数化）所使用的宏这个习惯语法的产物。它们至今仍然保留了类似的设计语义，尽管它们已经被更大程度地整合（集成）到编程语言中。C++模板已经很成熟，它们可以支持非常惯用的设计表达。第 8 章探究了作为表达领域之间依赖性的一种自然方式的这些更高级的结构。

共同性：结构。模板共享相同的结构。这包括代码结构和类模板的数据结构。函数模板是具有相近形式的各个算法的类似结构。我们将函数模板（本节的后面讲述）与类模板分开处理，这是因为应用对它们的处理大不相同。

差异性：详细的类型或值。模板带有调制和添加到代码生成的参数。模板参数通常不用于大型的结构变化。

模板可以为其每个差异参数定义一个默认值。正如第 3.6 节所述，默认值是多范型设计的一个重要部件。

模板特化

一个模板可以定义一个族。各个族成员可以用捕获了差异参数的适当的模板参数生成。使用“模板特化”（template specialization）是生成其他各种形式的一种方式。我们来

考虑一个带有参数T、名为Stack的模板：

```
template <class T> class Stack { . . . . };
```

我们可以为特定差异参数（这里是为整型的堆栈）规定此模板的一个特殊形式：

```
template<> class Stack<int> { . . . . };
```

这是消极差异性的相关形式，我们将在第6.11节深入讨论。

绑定：源码（编写）时间。模板抽象将依据代码中显式的差异参数，在编译时间或连接一编辑时间捆绑到它们的客户端。模板参数可以设置默认值：

```
template <class T, class SequenceClass = dequeue<T> >
class Stack {
  . . . .
};
```

例子：集合（比如前面的Stack）是模板的一般例子。

6.6 函数模板

函数模板和类模板都建立在相同的C++语言机制上。然而，因为它们一个支持数据抽象，一个支持过程抽象，所以二者有不同的设计用途。

共同性：结构。模板共享相同的源代码结构。函数模板是一个带细节（比如类型依赖性、常量、默认值等，最后都归结到差异参数中去）的算法抽象。

差异性：详细的类型和值。这与类模板的大致相同。

绑定：源码时间。模板抽象在编译时间或连接一编辑时间捆绑到它们的客户端。在绑定发生的代码中的差异参数是显式而直接的。

例子：函数sort是模板函数的一个典型例子，其参数是一个模板类型。模板扩展到适合于sort被调用时所带的参数类型的代码中：

```
template<class T> int sort(T[] elements, int size);
```

6.7 继承

程序员通常将继承当成一种代码复用的机制。说得更明确一点，就是C++继承可以同时表达子类型的导出和代码共同性。多范型设计不倾向于强调代码复用的角度，而是聚焦在（并细致区分）代码和行为共同性上。

共同性：行为和数据结构。在面向对象的设计和面向对象的编程中继承表现得都很

突出。在 C++尚处于初期时，继承的工业模型比今天所使用的模型要简单得多。当代的模型将行为的继承和结构的继承分离开来。换句话说，共同性有两个单独的组件：行为中的共同性和表达中的共同性。

在进行应用领域分析时，我们聚焦在行为的继承上。如果聚焦在实现结构中的共同性上，我们就可以增强设计的灵活性。基于实现结构的设计可以不延伸到相关的领域，甚至可以不延伸到原应用领域中新的应用。聚焦在行为上有助于为设计留下更大的灵活性。这个基本原则是诸如 CRC 卡[Beck1993]和职责驱动的设计[Wirfs-Brock1990]这些设计方法的核心。

例子： Number 层次是过去说明这一点的一个常用例子[①]。我们来考虑图 6.2 所示的设计。当代的很多面向对象的设计方法的第一步都是建立类行为，而不会过多地考虑实现。图 6.2（a）中的继承树（最抽象的类在顶端，最细化的类在底端）恰当地表达了一个行为的层次。对 Number 的运算是对 Complex 的运算的一个子集，而对 Complex 的运算又是对 Integer 的运算的一个子集。每个类都继承了层次结构中其上层的类的运算。这是应用领域中的类型的继承，也就是说，是行为的继承。除了后来出现的 C++实现，我们还可以用诸如 UML[Fowler+1997]、OMT[Rumbaugh1991]、Booch[Booch1994]或者 ROOM[Selic+1994]这样的设计表示法来画出这样一张图。

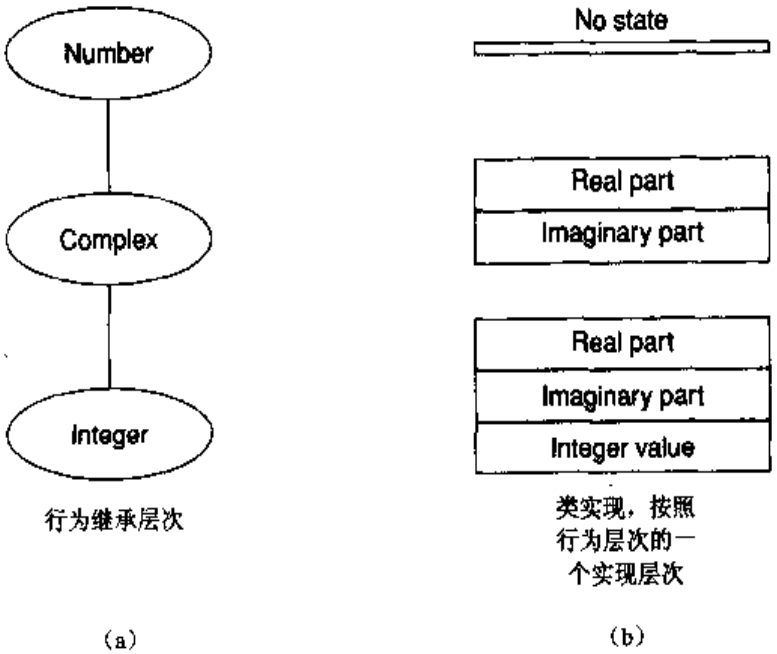


图 6.2 Number 的一个统一的子类型/实现继承层次

① 因为尽管 Complex 在除法下是闭合的，但其派生类 Integer 的除法并不闭合，所以有些设计者反对这个例子。然而，闭合是不需要被继承的。没有任何障碍阻止 Integer 的反转和除法运算返回 Complex 值。

6.7.1 结合领域

设计的很重要的一步就是将应用领域结构与方案领域结构结合起来。在 C++中，这通常意味着直接将 Rumbaugh、Booch 或 ROOM 类翻译成 C++类。假设我们从图 6.2（a）的行为层次上来派生出 C++类。我们为 Complex 的实部和虚部赋予相应的值字段，为类 Integer 赋予一个简单的整型字段。但因为 Integer 是从 Complex 派生而来，所以它继承了 Complex 在 C++继承层次结构中的所有属性，包括实部和虚部数据项。所以如果机器每 4 字节内存中同时出现了浮点和整型的数，那么 sizeof（Integer）可能是 12 字节。为提高效率（同时因为这违反了“将意外降到最低的原则”——Integer 包含了实部和虚部），我们希望 Integer 不继承这些数据字段。

如果结构和行为具有不同的继承层次结构时，设计者必须特别注意。C++可以通过私有继承和带公有继承的行为继承来捕获实现继承。私有继承包含了封装的语义（有些人称之为“实现层次”）。从私有基类派生出来的类的成员函数常常会调用它们的基类的成员函数，有时这些成员函数会将其语义交由它们的基类对应的成员函数决定。所以 Set 可以复用其私有基类 List 的实现，如图 6.3 所示。Set 无偿获得了 List 的所有方法，包括其算法和实现。这对函数（比如 Set<T>::add）很有帮助，它维持了元素的惟一性。派生类直接复用的惟一行为是 size 运算，继承也只传递了实现和结构。

```
template <class T>
class List: public Collection<T>
{
public:
     void put(const T&);
     T get();
     bool hasElement(const T &t)
          const;
     int size() const;
     . . . .
private:
     T *vec;
     int index;
};

void List<T>::put(const T &e)
{
     vec[index++] = e;
}
```

```
template <class T>
class Set: private List<T>
{
public:
     void add(const T&);
     T get();
     void remove(const T&);
     bool exists(const T&)
          const;
     using List<T>::size;
     . . . .
};


void Set<T>::add(const T &e)
{
     if (!exists(e))
          List<T>::put(e);
}
```

图 6.3 通过复用实现和结构继承，而不是行为

尽管公有继承表达了子类型的导出，但它也在派生类中生成了基类结构和实现的“逻辑拷贝”。程序员常可以根据需要使用此拷贝。例如，所有的 Window 抽象（大小、位置等）所共同的数据对所有的派生类也有意义。对派生类中的基类表示法的继承减轻了派生类的编写者重建共同实现结构的负担。但正如我们在图 6.2 中的 Number 的例子中所看到的，结构中的共同性并非总是遵循行为中的共同性。因为有了继承这个表示 C++中的两种类型的共同性和差异性（行为和结构）的机制，所以如果行为和结构共同性没有相互结合起来，那么我们就需要额外的机制。

这个问题的解决方案之一就是求助于一种可以支持行为和结构抽象的语言，正如 Eiffel[Meyer1992]和 Java[Flanagan1997]中所找到的语言。设计模式是另一种解决方案。Gamma、Helm、Vlissides 和 Johnson 在他们的模式著作[Gamma1995]中详尽地描述了这种模式。[Gamma1995]的第 4 章中的“类结构的模式”（class structural pattern）就阐明了这个问题。特别地，**Bridge** 模式直接解决了 Number 的问题，如图 6.4 所示。程序员为接口和实现分别创建了继承层次结构。来自这些层次的实例被“桥”从接口类对象（左侧）到实现类对象（右侧）结成对。这与[Coplien1992]的 envelope/letter 习惯用语类似。

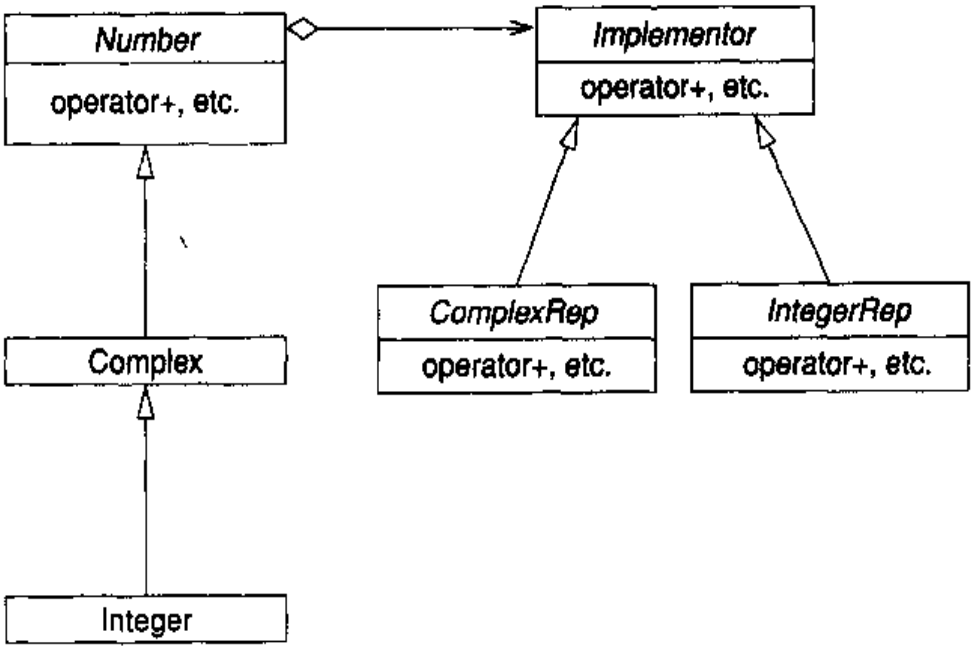


图 6.4 Number 的 **Bridge**（桥）模式，改编自[Gamma1995]

这种模式没有在 C++中偶然发现固有的表达法，尽管一种编程语言没有理由无法表达这种丰富的共同性和差异性。即使 C++语言不能以闭合的形式表达这些模式，我们仍然可以考虑将它们作为一种语言或实现技术。第 9 章更加深入地分析了多范型开发和模式。如果没有这些模式，我们就必须坚持用 C++的继承模型。

因为私有继承的语义是 **HAS-A** 而非 **IS-A**，所以大部分程序员（尤其是纯化论者）更愿意明确地使用封装而非私有继承，如图 6.5 所示。

```
template <class T>
class Set
{
public:
     void add(const T&);
     T empty() const;
     void remove(const T&);
     bool exists(const T&) const;
     int size() const;
     . . . .
private:
     List<T> theList;
};
```

```
template <class T>
int Set<T>::size() const {
     //   extra function call
     //   beyond what is used
     //   with private
     //   inheritance
     return theList.size();
}

template <class T>
void Set<T>::add(const T &e)
{
     if (!exists(e))
        theList.push_front(e);
}
```

图 6.5 为复用实现和结构（非行为）而进行的封装

如果类同时参与了子类型的导出层次和实现层次，则可以将公有继承用于子类型的导出，同时将私有继承用于实现。这可以通过对每种 **Adapter** 模式使用多重继承来实现，如图 6.6 所示。更一般的选择是对每种 **Bridge** 模式使用一个间接的额外层次，如图 6.7 所示。

```
class RealRep {
friend class Real;
friend class Complex;
     double rpart;
     . . . .
};

class ImaginaryRep {
friend class Complex;
     double ipart;
     . . . .
};
```

```
class Complex: public Number,
     private ComplexRep,
     private ImaginaryRep {
public:
     . . . .
};

class Real: public Number,
     private RealRep {
public:
     . . . .
};
```

图 6.6 对实现和接口的多重继承

差异性：结构或算法。C++继承不可拆分地同时表达了行为和结构中的共同性和差异性。我们可以使用继承来表达结合在一起的共同性和差异性，也可以在共同性中没有差异性时单独表达共同性。

```
class NumberRep {
    . . . .
};

class RealRep: NumberRep {
friend class Real;
friend class Complex;
    double rpart;
    . . . .
};

class ComplexRep: NumberRep {
friend class Complex;
    double ipart;
    . . . .
};
```

```
class Complex: public Number {
protected:
    NumberRep *rep;
    Complex (NumberRep *r):
        rep(r) { }
public:
    Complex():
        rep(new ComplexRep) { }
    . . . .
};

class Real: public Complex {
public:
    Real(): Complex
        (new RealRep) {
        . . . .
    }
    . . . .
};
```

图 6.7 使用 **Bridge** 的分离实现和子类型的层次结构

差异性分析存在一个有趣的细节。一个类可以展示来自共同性分析期间所建立的族中的其他类的积极差异性（参见第 3.3 节）。积极差异性越强，类就被从类继承层次结构的顶点（往下）推得越深。这与子类型的定义是一致的，并与 Liskov 的子代换原则[Liskov1988]保持了高度的一致。例如，OrderedCollection 可以支持一个 sort（排序）运算，而其基类 Collection 却不支持。并且，从 OrderedCollection 派生出来的 Set<int> 可以支持对自己的所有元素求和 sort（排序）的运算，而这些运算在接近这个层次结构的顶端时就消失了。

如果某个类显示了对比族中其他成员的足够强的消极差异性，那么它可能就不属于这个族。一个基于类型中的共同性的族中的消极差异性对应着带有相消的继承，这是通常应当避免的（[Coplien1992]，237~239）。在面向对象编程新手的工作中常常可以找到这个问题的经典例子，他们会从 List 中派生出 Set（或反向的派生）。因为 List 可以进行排序，所以它不能作为 Set 的基类（Set 是不能进行排序的，因为它没有排序的概念）。又因为 Set 确保它们的成员是惟一的，而 List 没有，所以 Set 也不能作为 List 的基类。二者都应当从一个更通用的类 Collection 派生而来。

绑定：编译时间。C++继承层次结构体系在编译时间被捆绑（请参见下一节）。

6.8 虚函数

虚函数在一个类层次结构内部形成了多个族，并且为继承的类之间的共同性形成了

大量的基准。它们是支持 ADTs 的主要 C++机制。

共同性：名字和行为。可以用两种方式对虚函数进行分组。第一种方式，将函数分组到实现应用领域的抽象数据类型的类中。第二种方式，将函数从不同的类分组到单个的继承层次结构体系中。类层次结构中的分组捕获了层次结构的函数和领域之间的共同关系，以及它们的更明显的“签名”（signature）的共同性。“签名”描述了一个函数的名字和参数类型。相关的虚函数都有相同的名字，并适用于类似对象（它们的类形成了一个继承层次结构体系）。

虚函数的一个族的成员通常共享相同的签名。但派生类签名中的返回类型可以比对应的基类签名中的返回类型的限制要少一些，而派生类签名中的成员函数的正式参数可以比对应基类签名中的成员函数的正式参数的限制要多一些。派生类成员函数的返回类型至少必须支持基类返回类型所支持的成员函数。基类成员函数的正式参数至少必须支持派生类正式参数所支持的这些成员函数。

简言之，虚函数共享了外部可观测的行为或共有的含义。例如，在类 Shape 下形成了一个继承层次结构体系的类均共享的成员函数 draw 及其他很多成员函数。所有的图形都共享“可以被绘制”这样一种“行为”，在这一点上它们就像重载的函数。从设计的角度来看，重载和虚函数之间的不同是：虚函数依赖于更大型的抽象（一个类），每个族成员（各个虚函数实例）在不同的类中，这些类通过继承相关。重载的函数很少明显依赖于其他抽象或设计关系。

差异性：算法实现。尽管虚函数共享相同的行为，但每个虚函数都以自己适合于其所属的类的方式来实现各自的行为。类 Circle 与类 Rectangle 的成员函数 draw 当然是不同的。将这种差异性看作一种实现细节通常是很有用的。所以虚函数是表达一个行为（对几个相关类是共同的）的实现差异性的一种结构。

继承使程序员能够处理相关的算法的差异性。如果集中算法各不相同，它们就可以打包到一个派生类中。这个派生类自身可以是一个管理的单位，此单位将成为概念上的差异参数的更高层次的值。例如，我们可以定义一个类，为故障恢复的三个层次实现相应的行为，它的派生类可以为一个多种的恢复策略而实现。需要故障恢复支持的类（或子系统）会将这个对象（或类）当作一个差异参数。用这种方法解析算法差异的继承可以与诸如 **Strategy** 模式（参见第 9.2.4 小节）这样的方法形成对比，后者可以调整以每次处理一个函数。

绑定：运行时间。既然虚函数族中的所有函数都共享相同的名字和相同的类层次结构，那么一个程序是如何从虚函数族中进行选择的？此选择是在运行时间根据对象（函数调用的对象）的类型而进行的。

例子：图像包中的图形是通过继承产生关联的可互换的族成员的一个经典例子。

6.9 共同性分析和多态性

“多态性”（polymorphism）意味着“很多种形式”。如果调整一下我们的共同性和差异性的角度，我们也可以说多态性意味着对几种相关的形式具有共性但每种形式又相互区别。C++支持几种形式的多态性，它们分别是前面五节的标题：

- 重载（overloading）
- 类模板（class templates）
- 函数模板（function templates）
- 继承（inheritance）
- 带虚函数的继承[①]（inheritance with virtual functions）

Wegner[Wegner1987]将多态性自身描述为被结构化地分成两组（“普遍的”和“特别的”）的四个成员的一个族（如图6.8所示）。在Wegner的模型中，对象是所关注的制品。“重载”（Overloading）描述了名字和语义相同但操作不同类型的对象的有关函数的族。“类型转换”（Casting）将一种对象类型转换成另一种类型。例如，我们可以在短整型和长整型之间进行转换。“参数多态性”（Parametric polymorphism）意味着以单一闭合形式描述的抽象的差异性可以通过一个参数列表来控制。“夹杂多态性”（inclusion polymorphism）描述了相互包含的抽象的集合层次结构，其中的所有抽象都可以依据最密闭集合的属性来处理。这就是我们对C++继承层次结构体系和虚函数的认识。

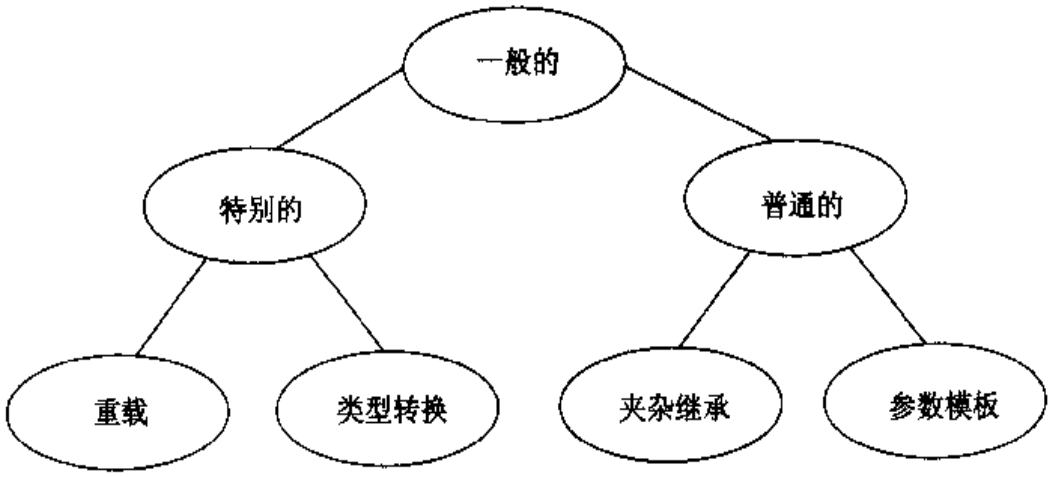


图6.8 Wegner多态性分类法

① 原文如此。——编者注

6.10 处理器指令

处理器指令（比如#ifdef、#ifndef、#if，以及终结的#endif）通常用于程序的细微改变。因为它们是处理器指令，所以可以很好地适用于数据结构和算法（代码结构）。尽管这些处理器指令被定义成C++语言的一部分，但很多程序员并没有深入全面地考虑这些优秀的语言特性。这是因为它们与其他的语言特性的整合相当弱，而且通常缺乏支持它们的符号工具。对于它们应当有选择地使用和慎用。

继承与#ifdef

注意：继承（第6.7节）也可以用于数据结构和算法结构中的细微改变。那么，它与#ifdef 有何不同？处理器指令可以高效地处理程序中最细微层次的粒度。继承可以处理细微的数据粒度，但这通常会带来一定的开销（例如，来自 C++数据类型对齐约束的开销）。函数是继承下的代码粒度单位。处理器指令可以处理任意细微的代码粒度。

有些设计模式通过建立在继承和虚函数上来管理任意细微的代码粒度。这使得拥有运行时绑定的任意细微的代码差异性成为可能。灵活性会带来一定的效率损失，参见第9章。

处理器指令在表达消极差异性时最有用，即对规则的例外。当一行代码在大多数时间必须出现，而在一两种情况下不出现时（出现与否在编译时才能知道），此时处理器指令通常是可行的解决方案。第3.3.2小节已经详细讨论了消极差异性，下一节也将对其进行讨论。

6.11 消极差异性

第 3.3 节已经介绍过积极和消极差异性。在应用分析期间寻找消极差异性是很重要的。在应用分析中找到以后，我们还必须了解如何在实现中捕获消极差异性。在C++中，我们可以使用通常为微调和实现细节所保留的方法来表达消极差异性。这些方法包括模板特化、设定参数默认值、间接寻址（为了取消数据成员）、私有继承（为了取消行为）和#ifdef。

我们可以认为 C++的每种范型都具有一个差异的规则。对于“对象范型”来说，差异的规则就是：族成员可以向在基类中捕获的共同性中添加新的函数或数据。多数范型的差异性没有涉及到共同性。消极差异性通过处理基础的共同性而违反了差异的规则——它们是这些规则的例外情况。

6.11.1 决定何时使用消极差异性

对于积极差异性，我们根据基础的共同性和差异性的本性来选择适当的 C++语言特性。对消极差异性的处理我们采用同样的方式。在差异性分析期间，我们怎么知道是否要使用积极差异性或消极差异性？

- 如果一个差异参数可以只在不影响基本的共同性的方式下变化，则使用积极差异性。例如，一个函数可对其所有的应用有同样的逻辑，但对于不同的应用，它可能需要使用不同的类型声明。模板函数将是捕获积极差异性的一种方法。
- 如果差异参数的某个值的范围带来违背了整体共同性的结构，则使用消极差异性。例如，如果某个函数需要针对某些目标平台集合有一些细微的不同，则使用#ifdef来区分这种不同。
- 有时差异性比共同性要大。此时，就需要对设计重新进行分析，将共同性和差异性对调过来。例如，某个函数的函数体必须要在几种不同的平台上存在重大差别。假设这种差别为消极差异性，我们使用#ifdef 来区分这些不同。如果想要在运行时间进行绑定，我们就应当使用简单的条件句而非#ifdef。如果想要在源码时间进行绑定，我们就应当捕获重载函数（可能添加了一个附加的函数参数作为差异参数）族中的共同名字和语义，以便实现可以自由地改变。

一个模板的例子

我们考虑将抽象 Stack（堆栈）作为一个简单的例子。大部分的 Stack 共享类似的数据结构，如图 6.9 所示。这个实现是非常通用的。它并没要求压入堆栈的条目（内容）有一个默认的构造函数来处理多态的数据类型，并且堆栈可以按所需要的规模轻易地变大。但此实现大大增加了内存的开销（被它所管理的对象消耗），尤其当所管理的对象很小的时候。我们可以为某些 Stack 采用一个更高效的实现；比如对于 Stack<int>，我们使用一个简单的矢量，如图 6.10 所示。如果知道自己需要一个整型堆栈，那么我们就不需要支持多态性的通则，我们可以消除所有这些指针的数据结构开销。如果堆栈变大了，则我们将为获得这种效率而付出一定的代价，但这对大多数实现来说还是可以接受的。

通过共同性分析我们发现，多数 Stack 共享相同的结构、相同的代码结构和相同的外部行为。我们来看差异参数 **Element Data Type**（堆栈所存储的元素的类型）来计算它是否违背了任何的共同性。Stack<int>违背了结构共同性和部分代码结构的共同性。如果确定这个异常只对差异参数 **Element Data Type** 的很少一部分选择产生影响，那么我们就可以将其当作消极差异性处理。

```
template <class T>
class Stack {
    StackRep<T> *rep;
    . . . .
};
```

```
template <class T>
class StackRep {
friend class Stack<T>;
    StackRep<T> *next;
    T *rep;
};
```

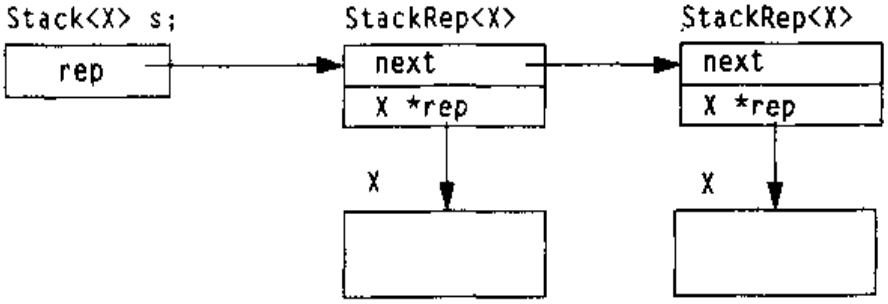


图 6.9 共同 Stack 数据结构

```
template<> class Stack<int> {
    int *rep;
    . . . .
};
```

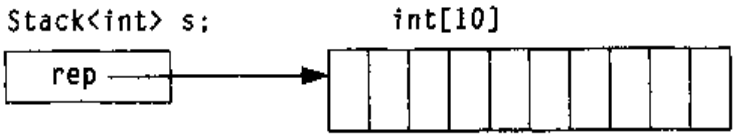


图 6.10 一个简单的堆栈

我们根据共同性和差异性来选择一个适当的 C++语言结构。特定的共同性和积极差异性惟一地选定了一个 C++结构。对于相应的消极差异性，我们选择通常与关联的积极差异性有关的一个 C++语言结构。对于暗示将模板作为一种实现方法的共同性和积极差异性，我们为关联的消极差异性使用模板特化。这就是我们为 Stack<int>所使用的方法。

一个默认参数设置的例子

默认参数设置是另一种可以表达一种消极差异形式的结构。现在来考虑这种情况下我们该怎么办：如果算法中存在主要的共性，而这种算法所使用的值中存在很小的差异。

例如，假设我们有一个应用需要模拟化学反应。某些分子有一个名为 Hydrate（氢氧化合物）的成员函数，但我们保留在非寻常的环境中使用“重水”（heavy water）的选择。我们将声明函数：

```
void Molecule::hydrate(hydrationMolecule = Water)
    { . . . . }
```

我们可以使用下面的任意一种形式来调用上面的函数：

```
Molecule *m;
. . . .
m->hydrate();    // 普通用法
m->hydrate(HeavyWater);    // 消极差异性
```

一个数据取消的例子

假设我们正构建一个文本编辑器，并且我们决定使用一个文本缓冲器来存储需要编辑的文本。这个类可能像下面这样：

```
template <class CharSet>
class TextBuffer {
public:
    Line<CharSet> getLine(const LineNumber &);
    . . . .
private:
    LineNumber firstLineInMemory, lastLineInMemory;
    int currentLineCapacity;
    . . . .
};
```

TextBuffer 可能有几个有关的派生类：

- PagedTextBuffer，它保留了主内存中的行工作区。
- LazyTextBuffer，它根据需要读取行中的内容，并一直保留到这些行被写出为止。
- SimpleTextBuffer，它总是在主内存中保留整个文件图像。

这些类可以使用直接的继承。但假设我们要引入一个 SingleLineTextBuffer，我们可以使用它来编辑命令行，或用于被编辑器实例化的对话框。如果我们从 TextBuffer 派生出 SingleLineTextBuffer，它将继承很多根本用不上的数据成员（或者以一种退化的方式使用一次），比如 firstLineInMemory、lastLineInMemory 和 currentLineCapacity 都置为 1 的情况。我们不能“取消”派生类中的基类成员函数。我们可以通过这种方式处理这个问题，即同时将基类和派生类中的数据析出到通过一个指针维持原抽象关系的单独结构中。首先，考虑诸如一个文本编辑器应用中，用于 **TextBuffer** 族的实现的类层次结构（下面的类是柄/体对中的体类对象）：

```
template <class CharSet>
class TextBufferRep {
public:
    void appendChar(const CharSet&);
    . . . .
};
```

```
template <class CharSet>
class PagedTextBufferRep:
    public TextBufferRep<CharSet>
{
    friend class PagedTextBuffer<CharSet>;
    LineNumber firstLineInMemory, lastLineInMemory;
    int currentLineCapacity;
    . . . .
};

template <class CharSet>
class LineTextBufferRep:
    public TextBufferRep<CharSet>
{
    . . . .
};
```

下面的类是相应的柄类，它代表了用户接口（实现正是通过用户接口管理的）。基类TextBuffer中的共同rep指针指向体类，声明为一个指向(通用)类型TextBufferRep<CharSet>的指针：

```
template <class CharSet>
class TextBuffer
{
public:
    Line<CharSet> getLine(const LineNumber &);
    . . . .
protected:
    TextBufferRep<CharSet> *rep;
    TextBuffer(TextBufferRep<CharSet>*);
    ~TextBuffer();
private:
    // encourage heap allocation - disallow instance copies
    TextBuffer(const TextBuffer<CharSet> &) { }
};
```

下面的每个类都继承了rep指针，但每个类都可以选择将此指针指向适合各自角色的相应TexBufferRep的变量。

```
template <class CharSet>
class PagedTextBuffer: public TextBuffer<CharSet>
{
public:
    PagedTextBuffer():
        TextBuffer<CharSet>
```

```
            (new PagedTextBufferRep<CharSet>) {
        . . . .
    }
    . . . .
};

template <class CharSet>
class LineTextBuffer: public TextBuffer<CharSet>
{
public:
    LineTextBuffer():
        TextBuffer<CharSet>(new
            LineTextBufferRep<CharSet>) {
        . . . .
    }
    . . . .
};
```

第6.7节中作为教学用例的Complex可以从消极差异性的角度来考虑。我们来考虑Complex层次结构的设计中这种直接的（但可能并不现实的）企图：

```
class Complex: public Number{
public:
    . . . .
private:
    double realPart, imaginaryPart;
};

class Imaginary: public Complex {
public:
    . . . .
private:
    ?
};

class Real: public Complex {
public:
    . . . .
private:
    ?
};
```

我们要取消派生类Imaginary中Complex的数据成员imaginaryPart。最灵活的解决方案是使用Bridge模式[Gamma1995]将数据表示法解析到通过一个共同基类结构关联

的两个结构中，如图 6.7 所示。从多范型设计的角度来看，我们可以将其看作是一种特殊种类的消极差异性，它保留行为中而没有保留结构中的共同性，或者保留了结构中而没有保留行为中的共同性。本书第 9 章深入介绍了多范型设计和模式之间的联系。

行为取消的一个例子

行为取消（通常称为“带取消的继承”）是面向对象设计中的消极差异性的最声名狼藉的例子。设计新手有时会认为 C++允许取消派生类中的一个虚函数。C++类型系统并不允许这样做，因此就使它能在编译时检查是否违反了子类型的可替换性。使用语境中（其中要求有一个基类的实例）的一个派生类的实例是可能的。这称为“Liskov 替换性原则”[Liskov1988]。

在使用了公有继承并且在公有继承的类中不允许行为的取消时，C++类型系统支持 Liskov 替换性。我们可以用通过使用私有继承来表达带取消的继承——行为的取消。当然，如果这么做，我们就丧失了在基类语境中使用派生类实例的权利（编译器确保了基类语境中不会出现私有派生的类对象）。然而，我们可以创建一个新的抽象，其功能性几乎与基类相同，但带有更少的具体操作。在第 6.7 节中，我们将这看作是不继承行为而复用实现的一种方法。这里我们想选择性地复用行为和实现中的某些部分：

```
template <class T>
class List {
public:
    T front() const;
    T back() const;
    int size() const;
    bool exists(const T&) const;
    void insert(const T&);
    . . . .
};
template <class T>
class Set: private List<T> {
public:
    using List<T>::size;    // same--uses base class
                            // implementation
    using List<T>::exists; // same--uses base class impl.
    void insert(const T&); // different--check for
                            // uniqueness
    . . . .
private:
    // class qualification here is just for documentation
```

```
    using List<T>::front;
    using List<T>::back;
};
```

行为取消的良好应用难得一见。一般总是由“重解析”（refactor）设计来替代（参见第6.11.2小节）。

逆变性的一个例子

逆变性是基类成员函数和覆盖它的派生类成员函数的参数类型之间的关系。派生类成员函数的参数通常与对应的基类成员函数中的参数是相同的。我们考虑这样一种情况，其中派生类参数比基类中的参数限制性更强：

```
class GIFImage{
// a class for general GIF images
    . . . .
};

class InterlacedGIFImage: public GIFImage {
// a class specialized for interlaced GIF images
public:
    . . . .
};

class GIF89aImage: public GIFImage {
// a specific format of GIF images
public:
    . . . .
};
class Window {
// a general window abstraction
public:
    . . . .
    virtual void draw(const GIFImage &, Point loc);
    . . . .
};

class HTTPWindow: public Window {
// a proxy object for a Web window
public:
    . . . .
    void draw(const InterlacedGIFImage &, Point loc);
    . . . .
};
```

现在来考虑下面的代码：

```
Window *newWindow = new HTTPWindow;
GIF89aImage picture;
. . . .
newWindow->draw(picture, loc);     // woops
```

假设最后一行调用 HTTPWindow::Draw。该函数需要一个至少可以支持 InterlacedGIFImage 的成员函数的参数，并且类型 GIF98aImage 的实参出现了一个问题。当派生类比基类中的成员函数参数的限制性更强时，C++通过“短路”（short-circuit）虚函数机制来解决这个问题。这样，Window::draw 将在前面的代码段中的最后一行被调用。

逆变性可以被当成是一个带取消的继承特例，它通常用一种模式（比如 **Visitor**[Gamma1995]）或通过重解析来表示：

```
class Window {
public:
    // handles GIF & GIF89a
    virtual void draw(const GIFImage &);

    // handles InterlacedGIFImages. In the base class
    // default behavior we do no interlacing, but derived
    // classes can override this behavior
    virtual void drawInterlace(const InterlacedGIFImage
        &image) {draw(image);}
    };
class HTTPWindow: public Window {
public:
    // perhaps a special version for GIFs on a Web window
    void draw(const GIFImage &);

    // interlace interlaceable GIFs on a Web window
    void drawInterlace(const InterlacedGIFImage &image) {
        // code for interlacing
        . . . .
    }
    . . . .
};
```

#ifdef的一个例子

Stroustrup 在 C++上要取得有成效的目标之一就是去掉 C 预编译器。诸如 const 类型修饰符的语言特性和 inline（内联）函数证实了这一目标。C++唯独没有直接去掉的预处理器功能就是支持条件编译的这些功能：#ifdef、#ifndef、#if、#else 和

#endif。我们使用这些结构来选择性地向程序中引入算法和数据结构段。许多编译器都试图优化 if 语句，以便让 if 语句的开销不超过执行预处理器条件编译指令的开销。但是，if 语句只可用于算法结构，而不能用于代码结构，所以程序员常采用#ifdef 进行一些细微的改变。对代码的调试就是一个典型的例子：

```
#ifdef TRACE_ON
#define TRACE_INIT_CALL_COUNT \
  static int trace_call_count = 0
#define TRACE(F) trace_show(#f); \
  ++trace_call_count
#else
#define TRACE_INIT_CALL_COUNT
#define TRACE(F)
#endif

void f(void) {
    TRACE_INIT_CALL_COUNT;
    TRACE(f);
    . . . .
}
```

注意：在一个程序中不能对数据差异性使用预处理器指令。假设某个程序使用#ifdef，根据差异参数在两个选项之间进行选择，从而实现具有不同数据结构（例如，相同类的不同版本）的两种不同族成员。没有哪个单个的程序可以同时包含两种不同的族成员。C++依赖于程序中的数据布局的一致表示法。

6.11.2 消极差异性与领域拆分

前一节的方法适用于“小”的消极差异性。随着消极差异性变大，它已不再描述易变的特性：它变成了一个共同性，且它的实现变成了差异性。大型的消极差异性通常应当当作积极差异性来对待。

行为取消的一个例子

最常见的例子再次涉及到面向对象设计中带取消的继承的问题。我们可以将共同性归结到一个更高层次的基类中，而不是取消派生类中假定的基类共同性。重新分析前面的例子，我们引入一个（不是很实际的）基类 Collection：

```
template <class T>
class Collection {
public:
    int count() const;          // common to all derived
```

```
    bool has(const T&) const;  //      classes, so not
    // virtual. insert is deferred to the derived class if
    // one likes, one can declare a placeholder here, to
    // capture the fact that the behavior is common to all
    // family members
    virtual void insert(const T&) = 0;
    . . . .
};

template <class T>
class List: public Collection<T> {
public:
    T head() const;                 // peculiar to List;
    T tail() const;                 // this, too
    using Collection<T>::count;     // just use the
    using Collection<T>::has;       // base class
                                    // version
    void insert(const T&);          // simple list
                                    // insertion
    . . . .
};

template <class T>
class Set: public Collection<T> {
public:
    // no head, tail -- not meaningful for Sets
    using Collection<T>::count;     // just making the
                                    // default
    using Collection<T>::has;       // explicit (good
                                    // documentation)
    void insert(const T&);          // different--must
                                    // check
    . . . .                         // for uniqueness
};
```

也就是说，我们创建一个更宽泛的基类抽象 Collection，它是跨 List 和 Set 共同的，而不是使用相对狭窄的抽象 List 作为 Set 的共同性基类，将 Set 用消极差异性表达为 List 的一个变量。List 和 Set 都可以依据 Collection 只使用积极差异性来描述。

模板的一个例子

模板特化可以容纳实现代码和数据结构中不同种类的变化。有了这种灵活度，就没有必要得出与非特化的原来的实例共同性很少的模板的特殊情况了。例如，一个典型的矢量可能像这样：

```
template<class T, int size = 50>
class vector {
public:
    . . . .
    T operator[](unsigned int index)
        { return rep[index]; }
    . . . .
private:
    T *rep;
};
```

但专用的位矢量的实现可能看起来就完全不一样。我们用模板特化的一个直接应用来捕获这种差异：

```
unsigned short vectorMasks[16] = {
    0000001,
    . . . .
    0100000
};

template<int size = 50>
class vector<bool, size> {
public:
    . . . .
    bool operator[](unsigned int index) {
        return rep[index / (8 * sizeof(unsigned short))] &
        vectorMasks[i % (8 * sizeof(unsigned short))];
    }
    . . . .
private:
    unsigned short *rep;
};
```

数据取消的一个例子

前一节中（第 6.11.1 小节）的“数据取消的一个例子”中所描述的方法可以有意识地进行扩展，这样它就足以处理数据成员的独立取消和跨族成员完全不相关的数据格式。即使继承层次体系每一层的数据都是完全不同的，额外的间接层次也将容纳此差异性。

#ifdef的一个例子

有时差异性会留下几乎没有共同代码的单个过程——所有的代码都在不同的部分：

```
void f() {
#if C1
    . . . .
    // lots of code
```

```
#else
    . . . .
    // lots of code
#endif
}
```

条件编译结构几乎可用于表达任何差异性。使用它们应有所节制。其他的方法（比如解析贯穿继承的变量、函数重载，有时还有模板）常常比条件预编译器指令更加合适。有许多惯用的预编译器方法都只处在对语言关注的边缘地带。例如，在前面的情况中，我们可以创建两个完整独立的函数放在头文件中作为静态函数。要选择相应的函数，只需要#include 相应的头即可（可能使用一个#ifdef）。

6.11.3 消极差异性小结

表 6.1 总结了消极差异性和它是如何与用于表达共同性和差异性的 C++特性进行交互的。前两列反映了来自领域分析的共同性和极小的差异性。第三列回忆用于共同性/差异性对的 C++语言特性，最后一列为差异性的异常指明了补救方法。前三列在表 6.2 中详细展开。

表 6.1 为消极差异性选择 C++特性

共同性的种类	差异性的种类	针对积极差异性的 C++特性	针对消极差异性的 C++特性
名字和行为	跨一个参数类型或值所控制的结构或算法	模板	模板特化
结构、算法、名字和行为	细微结构、值或类型	模板	模板参数的默认值设置
（函数的）语义和名字	公式或算法中的默认值签名	函数参数的默认值设置重载	提供显式参数重载
一些数据结构中的共同性；也可能是算法中的共同性	数据结构中的成员	继承、添加数据成员	使用指针指向可选实现的重解析
结构和算法中的一些共同性	行为	继承、重载或添加虚函数	私有继承或 HAS-A 容器
大部分的源代码	细微算法	#ifdef	#ifdef
所有其他	所有其他	通常没有	参见第 9 章

表 6.2 C++编程语言领域的共同性和积极差异性

共同性	差异性	绑 定	例 化	C++机制
函数名和语义	除算法结构以外的其他任何差异性	源码时间	n/a	template
	细微的算法	编译时间	n/a	#ifdef
	细微或总的算法	编译时间	n/a	重载
所有的数据结构	状态值	运行时间	是	struct、简单类型
	一个小的值集合	运行时间	是	enum
(可选，相关运算)	类型、值和状态	源码时间	是	template
相关的运算和一些结构（积极差异性）	状态值	源码时间	否	模块（带静态成员的类）
	状态值	源码时间	是	struct、class
	数据结构和状态	编译时间	可选	继承
	算法（有其是多种）及（可选的）数据结构和状态	编译时间	可选	继承（与第 9.2.4 小节的策略对比）
	算法、（可选）数据结构和状态	运行时间	可选	虚函数

此表没有覆盖差异性“取代”和成为共同性的情况。这些应当根据情况按照第 6.11.2 小节中的分析进行解决。

6.12 C++方案分析小结：一个族列表

我们可以在表 6.2 中总结 C++方案领域，此表提供了语言特性如何相互关联的统一概观。在某一个层次上，此表总结了具体 C++语言特性的值。若程序员超出初级阶段，从一个设计的角度学习 C++，此表对他们来说将是一个很有用的工具。但在较高的层次，我们将这个表当作指导始于较高层次的分析和设计的设计决定的“虚拟机器”。有些语言特性看起来在高层次应用得比其他层次要好。例如，我们将模板和派生类当作强大的架构结构，同时却很少将条件编译指令和参数默认值的设置当作这种结构。

现在有了设计模型，我们就可以进入到下一章，使用这个模型来设计系统（使用多种范型）。

第 7 章
范型的简单混合

在本章中，我们将一个问题分成几个部分，以便每个部分都可以用单个的范型进行设计。这种“分而治之”的方法是多范型开发的最简单形式。

7.1 将所有范型放在一起：多范型设计概览

大部分的软件项目都需要多种范型。即使是最“纯”的面向对象的项目，基于代码的逻辑和共同的工程实践（比如循环优化）的考虑，我们也要再回过头来谈谈各种方法或成员函数中的过程思想。多种范型通常出现在复杂应用的最高层次上，模板、重载及其他的 C++语言特性可能作为首要的架构抽象出现。多范型设计可以帮助我们选择使用哪一种 C++范型。在前面的章节中，我们建立了共同性和差异性分析的基础，并将其作为多范型设计的构件块。现在是进入设计并创建一些抽象的决定性时候了。

在本节中，我们将简短地预览使用多范型设计的方法，并定义带领我们从共同性分析进入到实现的步骤。

7.1.1 不能以一适百

第 1 章启发了对多种范型的需求。即使安排了一个扩展的范型选项板（正如在 C++中我们所做的那样）我们仍然需要一些方法将这些范型组合到实现中。有多种不同的方法用来混合范型。最复杂的方法是将多个子领域的范型（如第 8 章所述）编织在一起的方法。如果我们正在构建一个作为复用模块（不会再分）的框架，那么由多范型设计带

来的内部耦合就通过不断增长的可表达性，以及框架中外部参数化和扩展的位置的简易演进进行调整。

对一个系统清晰地进行划分，以便各个范型不会横跨设计划分，这一点通常可以做到。在很多的设计层次上（甚至是架构的最高层上）都可做到这一点。有时，主要的系统块是GUI或数据库——实现范型与C++无关。我们将在第7.6节中研究这种“外部范型”。在更详细的实现层次中，我们可能会发现：有规律地结构化了的设计中有很大一部分都可以简单地用单个范型来处理。此范型通常是对象范型，有时是过程，有时是模块。尽管整体的设计可能需要几个范型才能完成，但我们通常实行“分而治之”，以便各个部分（“块”）都可以使用单一的范型及其方法。这是本章所研究的类似的子领域的共同情况。

设计的目标之一就是将各个部分之间的耦合减到最小，并将它们的内聚增到最大。在将领域划分成子领域时密切关注共同性可以帮助我们实现这些目标。我们通过对领域的族成员“沿着可在实现技术中自然地表达的这些共同性的轴线”进行分组，找到子领域，然后从简单的共同性分析入手。在共同性分析完成以后，我们就要确认是否可以用编程语言表达这些共同性。例如，如果发现族成员共享共同的数据结构，我们就要在C++中寻找表达这种共同的数据结构的方法，并找到切合需要的继承。如果发现族成员共享相同的接口，而每个成员对此接口的实现又不同，那么我们就可以使用带虚函数的继承。如果发现共同的代码结构和行为，没有发现接口中的差异性，我们就可以使用模板[①]。这个过程可以经常性地重复。

例如，我们来考虑矩阵和矢量的领域。因为所有的矩阵共享共同的行为，所以我们首先将所有的矩阵分在一组，所有的矢量也进行类似的分组。矢量和矩阵自身又都是一个子领域。我们也可能注意到函数族：乘法（矢量与矢量、矩阵与矢量、矩阵与矩阵、稀疏矩阵与矢量、矢量与对角矩阵等）、除法（同前）、加法（同前）和减法（同前）。每种基本运算都成为一个成员可以用重载来表达的子领域。

尽管耦合和内聚被用作设计或架构的原则，但它们直接为代码的质量带来的好处并不如为维护代码的人的工作质量带来的好处。好的去耦使团队或个人独立工作成为可能。紧密的耦合使得小团队难以独立地维护它们的代码，这就意味着不能局部地得出设计决定，这样将会影响生产率。软件去耦是手段，团队去耦是目标。如果政治、文化、策略或历史要求某个团队结构与软件考虑事项无关，那么软件结构就应当遵从组织，不得违反。[②]这就意味着最好的（实用）领域通常来自于一个直觉的市场或商业远景，而不是共

① 参见Scott Meyers的《Effective C++》[Meyers1992]中的一个类似的例子。

② 有关组织和结构之间关系的更多内容，请参考[Coplien1995a]中的“Organization Follows Location”和“Conway's Law”模式。

同性和差异性的领域分析原则。所以不仅任何单个范型应服从好的领域分析所驱动的多范型设计，而且多范型设计也应服从常识和主要的业务结构。没有哪种方法可以在任何时候都表现得很好。

容错系统中的审查领域就是一个例子。审查用于搜集数据以检测问题，并逐步增强恢复作用，从而将系统恢复到一个一致状态。简单的共同性分析就可以表明：关键的数据结构同时具有普通的遍历方法和审查方法，这些方法都涉及内部深层的数据结构。这里很可能存在共同性。差异性存在于遍历的样式中，我们可以设想用一个迭代来访问每个元素，依次应用依据某个参数（“审查”或“遍历”）所选择的函数。有时将这种方法称为“授权审查”（delegated audit），其中的审查自身并非一个领域，而是关键数据结构的职责。在很多这样的系统中，审查自身历来都是一个领域。似乎有很多技术上的参数支持将审查归为一个单独的领域。首先，有时必须相互审查数据结构。这就意味着需要一个独立的代理，来审查相对于系统状态的其他方面的多重数据结构。其次，审查会为审查、建立在审查专门领域知识基础之上的技巧、超出数据结构设计者的专业知识以外的技巧使用灵活的策略和技巧。这些技巧可能要依赖于基础的硬件或软件平台，最好将它们归到它们自身的领域中，而不要分散关于系统的认识。再次，审查需要像任何其他的系统活动一样确定时间进度，并且审查时间进度的确定需要其自身的控制轨迹。即使这些不是历史原因，我们也可以从历史发展中出现的针对特定生产线的优秀审查组织中获得一些线索，并遵循这种深厚的传统，而不是完全服从技术上的分析。经验规则表明，处理尚不完善的软件结构比变更组织容易。有时，这个经验规则指导了领域的系统阐述。

我们在谈论差异性时，共同性的背景通常是隐含的。这就是说，有时领域分析中的差异性的词汇提供了有关实现中的共同性的线索。实现中的差异性（假定是共同的行为）指向继承。例如，所有的矩阵（稀疏、单位、上下三角等）抽象的行为类似（都可以加上、乘以、或除以其他矩阵），但实现它们的运算和数据结构却是不同的。因此矩阵抽象之间应当存在着一个继承关系。当谈论函数签名中的差异时，我们就暗示了某种共同性。我们可以捕获这种共同性作为重载函数的一个族，这些函数都共享共同的名字（同时带有矩阵的代数运算）。

7.1.2 复杂程度

设计者面临着覆盖大范围的复杂性的问题。简单的设计问题可以直接进行编码，困难的设计问题就需要更多的规划、内省和分析。我们可以开发一个简单的复杂性模型（“所有的模型都是对有限信息的估算，很多模型都是有用的”）来说明多范型设计对设计的健

壮性有何贡献。

我们根据规模来考虑复杂性，或者将复杂性与代码、人数对应起来。但有些庞大的项目总体设计上很简单（卫星数据的大型数据库仍然使用已经过检验证实的数据库技术），很多项目在我们所说的复杂性上都是不同的。那么，是什么导致了项目变得很复杂？

我们使用抽象来表达复杂性。问题越复杂，我们就必须使用越多的抽象来理解它。有些项目看起来很容易分解成抽象。大部分设计方法使用分层求精，基于自上而下或自下而上的策略而构建。分层组织看起来已经深深扎根于西方人的思想中了，它支配了我们表达抽象和组织世界的方式。如果一个项目充分地使用了分层的求精，它实际上看起来就不复杂——我们可选择从某个任意层次的细节来审视它。

设计层次结构有很多种。第一种也是最重要的一种是过程分解——建立在其他算法上的算法。我们可以在块结构化设计中找到相关的但更丰富的层次，这种设计的主要结构遵循过程分解，但其设计规则却随同过程划分数据的作用域。基于对象的程序设计使用“实现层次结构”[Booch1994]来构建对象的层次结构。与基于对象的程序设计不同，面向对象的程序设计使用遵守共同接口的类的层次。

我们还发现了一些分层的范型（实际上很少）。它们都是很少被实践过的功能范型，服务于诸如 FP、ML 和 Tim Budd 的 **Leda**[Budd1995]这样的语言。但要记住，很多流行的范型（比如关系数据库[Date1986]）也不属于分层的模型，这一点很重要。

有些项目非常复杂，一般的抽象方法难以处理它们。导致这些项目变得复杂的原因是它们缺少“单一的”层次结构。复杂系统展示了很多相互关联的层次结构。如果我们要使用自上而下的方法来表达这些系统的抽象，我们就必须从很多“顶点”来分析它们。电信系统有呼叫处理顶点、记账顶点、维修顶点、发展和管理顶点、容错顶点以及其他顶点。这些顶点下面的抽象树之间“以恶性的方式”相互影响，这就使系统变得很复杂。“复杂性与系统的清晰、有意义的视图的数量成比例”。

多范型设计是如何适应这种应用分类的？我们可以以多种不同的方式使用多范型设计来处理不同类型的问题。这些问题分布在从单领域中单个范型的统一应用到跨多个混杂范型的多范型的应用区段。我们来考虑下列每种情况中的示例的有意义的视图数量。

单个领域、单个范型

这是最简单的情况：构建单个的产品族，其中的所有产品都遵循一组共同性和差异性类别（第 2.3 节介绍过了共同性和差异性类别）。

示例：现实世界中这样的例子不太多，因为世界一般都不会这么简单。但排序算法族确实是相互独立的，那么它们就可以归为此类。

多个去耦的子领域、单个范型

定性地看，全部使用相同的范型来设计带有多个领域的系统，这与单个领域单个范型的情况并没有什么不同。此项目仍然可以使用单个工具集和单个的练习程序。每个领域都有自身的与领域的业务所关注的内容有关的词汇表，但所有的抽象都是相同的“形状”。

示例：我们来考虑一个监视工业控制过程的软件。面向对象的设计可能会用于与工业实现交互的软件架构，因为对象范型通常很适合于物理设备。面向对象的设计也可能用于系统图形界面，它是同一个系统中的一个不同的领域。

多个去耦的子领域、每个子领域使用单个范型

这样一个项目可以作为多个独立的项目（各个项目之间通过明确定义的接口和机制进行交互）来运行。每个项目都有其自身的工具和设计方法。一个领域的软件可以依据其他领域的范型或通过一个全局一致的交互范型（过程、对象、消息、数据库等）与其他领域的软件进行通信。

示例：带有数据管理（数据库范型）和人员接口（对象范型）的生产线的存货跟踪系统就是一个例子。

这是本章我们要重点关注的一类问题。

多个去耦的子领域、每个子领域使用多个范型

很多项目都属于这一类别。多范型设计帮助我们识别出每个领域的差异参数，以及将差异参数转变成设计结构的范型。多范型设计没有规定如何将这些范型编织在一起。但设计维度通常是相互垂直的，有经验的C++设计者常可以预见到范型混合在一起的“明显”的设计。

示例：**File** 领域可能同时拥有作为差异参数的 **Character Set** 和 **Record Formatting**。**Character Set** 成为一个模板参数。**Record Formatting** 可以用继承（同时为所支持的数据结构和格式化那些来自和去往磁盘的结构的算法）来处理。此设计一般为：

```
template <class CharSet>
class File {
public:
    virtual int read(const CharSet*);
    . . . .
};

template <class CharSet>
class WindowsFile: public File<CharSet> {
public:
```

```
    int read(const CharSet*);
    . . . .
};
```

我们将在第8章中讲述这个复杂性层次。

有向非循环图（DAG）中的多个子领域、多个范型

一个领域可能要依赖于另外一个范型。如果依赖关系图包含循环，那么此图的行为就好像它具有无限深的依赖性。当子领域依赖性形成一个有向非循环图（Directed Acyclic Graph，DAG），问题就解决了：每个子领域都可以将其他领域当成一个模块、一个独立的单位。

示例：我们来考虑一个复杂一些的例子。**Parser**（分析程序）和 **Code Generation**（代码生成）领域都要依赖于 **Machine Architecture**（机器架构）领域。而机器架构可能又将 **Instruction Mappings** 和 **Word Characteristics** 作为差异参数。**Word Characteristics** 自身可能是一个可以被参数化为“小连杆头”（little-endian）和“大连杆头”（big-endian）的领域，并用字节表示长度，用特殊填充物和调整特性来表示原始的硬件类型。

循环子领域

子领域可能（并且通常）相互作为差异参数。我们来考虑经典的三个领域（MVC）：**Model**（模型）、**View**（视图）和 **Controller**（控制器）。Model 领域代表应用信息，比如其他客户端所关心的应用状态（通常是一个人与软件进行交互）。**View** 领域通常在一个人机接口上呈现出 **Model** 信息的抽象视图。**Controller** 领域将用户输入分配到 **Model** 和 **View** 领域中去。这些类都遵从一个简单的更新协议的集合，以保持 **View** 与 **Model** 的同步。例如，所有有关的 **View** 都自行注册到 **Model**（其数据是 **View** 所关心的）。所有的 **Model** 会在有关的应用状态发生变化时通知相关的 **View**。这些协议独立于应用语义，这就意味着 **Model**、**View** 或 **Controller** 都不可以依赖于另外二者的具体版本。但这三个领域经常受制于这种理想的去耦，因为每个领域通常都要依赖于另外的一个或两个领域的细节（元件）。

我们来考虑一个条目表应用[①]，它可能是电子表格，或者表格自身是一种特殊的视图。**EntryForm** 领域将 **Model** 的数据字段当作差异参数，此领域构建了一个表格，以便所有的用户都可以通过 **EntryForm** 视图来查看或更改这些数据字段。尽管此模型必须支持全部模型所具有的共同行为（比如在其内容发生变化时改变 **View**），但它不应当依赖于某个特定的 **EntryForm** 或控制器。**Controller** 依赖于特定的 **EntryForm** 的结构，因为它必须知道文本字段、按钮和滑杆出现在屏幕的什么地方，以便它可以对鼠标点击和键盘事

① 此例是一次看见 Alan Wills（Alan@cs.man.ac.uk）在万维网上所贴的 EntryForm 可插入视图而联想到的。

件进行合理的调度。**EntryForm** 依赖于 **Controller**。这是因为它可能获取引起交互加亮的鼠标输入和键盘输入，或者从单独的过程获取输入（比如监视器或调试过程），对这些内容就应当禁用加亮。这导致了标准 **MVC** 架构中派生类层次的耦合。通常，派生类只有通过它们的基类层次的接口进行耦合（如图 7.1 所示）。

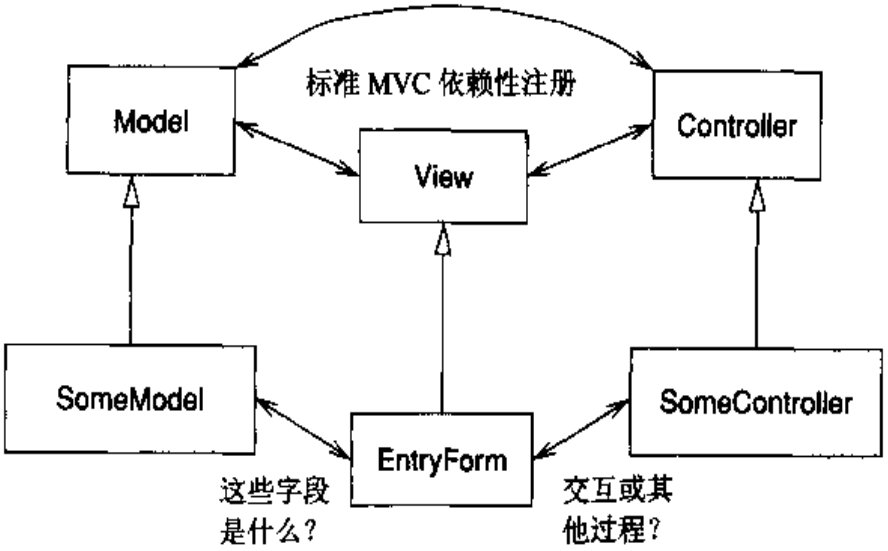


图 7.1 经典 MVC 中和更加耦合的 EntryForm 示例中的关系

这些依赖性与经典 MVC 架构相冲突，经典 MVC 结构的领域依赖性是在抽象基类接口中捕获的。尽管 **Controller** 依赖于抽象基类 View 的接口，但类 EntryForm（它在 **View** 领域中）依赖于 **Controller** 领域中的 SomeController。因为 **Controller** 依赖于 **View** 的结构（也就是依赖于 **EntryForm** 的结构），所以此设计违背了经典的 MVC 结构。由于 View（**EntryForm**）和 **Controller** 之间存在循环依赖性，所以此设计超出了简单多范型设计的范围。设计者常常通过将 **View** 和 **Controller** 这两个领域合并成一个领域来解决这一问题。这里，我们试着保持二者的分离，以便更好地理解二者之间的设计关系。

这种设计中的技巧是打破依赖性循环。我们可以对依赖性的一个方向使用一种耦合方法，而对依赖性的另一个方向使用另一种方法——通常是基于指针的方法。这里，我们可以将 **Controller** 当作一个提供了一个接口（在启动时，**EntryForm** 可通过它了解屏幕结构）的“可插入”类。**EntryForm** 可以通过所有的族成员通用的单个抽象基类接口与不同类型的 **Controller** 进行交互。

这种设计通常产生于应用类及其 I/O 接口之间的耦合。我们将对此种设计的详细分析推迟到第 8.4 节中进行。

7.2 多范型设计的活动

本书前面的章节介绍了多范型设计的构件块：共同性和差异性分析，应用领域和方案领域分析。现在，有了所有的构建块，我们就可以考虑如何将它们组合成一个设计策略了。

我们已经深入地学过了应用领域分析和方案领域分析。二者都是领域分析的具体应用。设计是将应用分析的结构与方案领域的可用结构结合起来的活动。这里，我们将这种活动称为“转变分析”（transformation analysis）。它是一种处理跨领域转变的分析活动——对映射结构的研究。

我们可以将整个设计方法中的这些方法组编成步骤。这至少有一些误导，因为设计很少会以可预见的步骤序列进行下去。这些步骤中有很多可以（也应当）并行地进行。人员配备和业务需求规定了当前是否要执行某些步骤（比如应用领域分析和方案领域分析）。有些步骤可以在项目开始的时候就已经完成（比如C++编程语言的一个领域分析），这样就有可能免除和缩减一些步骤。

对于任何一种方法来说，逐步地处理这些步骤是很危险的。团队应当只执行这些能够为他们带来共享的进步前景的设计活动。本书中的方法都是遵循形成和支持设计的原则。团队应当使用这些方法以获得更充分的进度，并用来支持项目和业务所特有的业务目标。一般来说，多范型设计过程的步骤如下：

1. 将问题分解成可以由直觉得到的子领域（第4.1.4小节）[①]。

一个有经验的设计者会运用领域经验形成新的系统架构。大部分的高层次的系统构造都是直觉性的。最初分割而成的“块”是独立的业务区域，每个这样的业务区域都有自己的历史和专门知识。（参见第7.1.1小节中关于此问题的讨论。）

一旦构建了全新的内容，我们就无法使用领域经验来分割形成最初的架构。此时，我们使用第4章中介绍的方法（参见后面的步骤3）来分析整个问题。设计团队可以在问题领域中寻找共同性，并使用这些共同性将问题划分成多个子领域。此活动可以帮助锻炼关于此领域的直觉。

本章着重讨论的是子领域基本上已经去耦的领域。对于一个好的子领域，所要考虑的主要事项可能是：它们是否可单独上市销售（第4.3.2小节）？如果领域是相互紧密耦合的，它们就可以结合形成一个更大的领域。如果市场力量和其他的设计考虑建议让紧密耦合的子领域保持各自的“身份”，那么就使用第8章所讲述的方法。如果在有关的领

① 本书有的地方直接称之为“直觉子领域”。——译者注

域模式中找到了合适的解决方案，那么这个模式（第 9 章）也可以用来处理这类问题。

2．前面处理过这样的设计吗？

在从零开始之前，我们想要复用已存在的设计。对尚不成熟和尚未被充分理解的领域，多范型设计是一个不错的选择。

3．分析每个应用子领域（第 4 章）。

多范型设计（尤其是共同性和差异性分析的活动）支持设计者和用户之间以及设计团队内的有效沟通。应用领域分析在某种程度上是市场活动，其任务是评估应用的市场宽度，并预期市场将如何发展。

并非所有的领域都能用多范型分析来处理。有些领域不需要（或很少需要）跨平台和生产线的成员的差异就可以实现。这些“单片领域”常常依附于传统的设计实践。我们在嵌入式系统中找到了这样的例子，比如过载控制、处理机管理和其他领域。共同性分析可以帮助设计者划分这些领域，但它们没有与我们在代表软件族的领域中发现的相同类型的“热点”——差异参数。有些领域难以同其他领域分离开来，这样设计者就必须使用诸如第 8 章所介绍的方法来管理它们之间的耦合。

大部分应用领域分析预见了解决方案可用的方法。这就意味着应用领域分析应当继续进行方案领域分析，或继续着眼于以前的方案领域分析。

一个项目可能会在逐次的发布中更新其应用领域分析。

4．分析方案领域（第 6 章）。

一个项目通常拥有多个方案领域。本书聚焦 C++作为一个方案领域，同时也包括了（在第 9 章中）一般的方案领域模式。所介绍的 C++和模式的方案领域分析活动都很彻底。对于 C++，请参照第 6 章介绍的内容。要预先考虑如何将应用分析和方案空间结合起来，并让自己熟悉相关的方案领域模式和相关的应用领域模式。如果将模式当成是一个单独类型的方案领域，我们预先考虑到：建立在过去的成功基础之上要好于自己重新开始。即使共同性分析是基于模式的设计的基础之一（参见第 9.1 节），模式也可能不让共同性和差异性变得明显。我们不应当强行将模式改写成共同性和差异性分析术语，应当按实际需要来使用它们。但有些模式“可以”在多范型设计框架中捕获。我们将在第 9 章深入讨论这一问题。

有些方案领域比其他方案领域更适合给定的问题领域。设计团队经常可以根据过去的经验来选择合适的解决方案的方法。例如，故障跟踪系统可以构建在数据库上，作为一个方案领域，也可以将 GUI 构造器作为另一个方案领域。多范型设计的这一步必须从对象或 C++领域中拿出解决方案。我们可以使用本书中介绍的方法来获得数据库和 GUI 方案。但本书只是浅层次地涉及所有可用的解决方案，例如，本书没有具体讨论关于数

据库和 GUI 构造器的任何具体内容。即使本书作了介绍，大规模使用正式的方法也是不必要的。多范型设计更适用于处理一种编程语言特有的方案领域，并支持这些交替使用语言特性的设计决定。

新领域中的设计问题更具挑战性。对新领域的应用领域分析常常会挑战设计者对合适的方案领域方法的直觉。这个合适的方法与其他的常用方法（比如第 6 章中介绍的这些方法）一起作为方案领域分析的“候选”方法。

设计者肯定很想对所有可用的解决方案方法进行综合分析，检验每种方法是否适合于某个应用领域分析。这种盲目的探索通常并不值得。我们应尽可能在经验的基础上，或在其他有经验的设计者的直觉的基础上进行设计分析。多范型设计成为这种直觉的审查者，并提供了多种方法和词汇表来规范设计。

5. 使用领域分析映射到可用的方案领域分析（第 7 章）。

这是多范型设计的核心，因为它处理了设计的主要问题之一：选择适合于问题的实现结构。我们分析每个应用领域的共同性类别、绑定时间、例化和默认值设置。这些参数都引导设计者得出解决方案的范型：对象、模板、模板特化、重载或其他范型。

转变分析通常指向类或继承（对象范型）作为相应方案领域的方法。除多范型设计以外，还有很多其他的工具、技术和方法，它们对对象范型的诠释比这里所描述的更加充分。如果多范型设计和经验表明它们适用于相关的领域，那么在这些主流方法的基础上进行设计分析是很明智的。

成熟的方案领域（比如客户端/服务器架构、人机接口设计、分布式处理和容错计算）的模式文化体（它描述一般问题的常用模式[PLoP1995]、[PLoP1996]、[PLoP1997]）在不断增长。模式可以通过提高设计复用的水平来缩短大量的设计努力。

任何单个的项目都可以使用多个方案领域方法。甚至还可以在单个的项目中使用多种编程语言。项目管理必须仔细对多种方案领域工具在技术上的适用性和实际问题（比如培训和工具支持）进行平衡。

有关这种方法的详细内容，请参考 Lai 和 Weiss[Weiss1999]的著作。

6. 设计使用面向应用的语言（AOL）的可能性。

有时，找到非常适合应用领域的任何实现领域方法都比较困难。领域的共同性和差异性可能不适合任何已存在的方案领域结构。此时，最好选择为这种问题构建一个“定制的”方案领域结构，也就是说，专门为此领域构建一种新的语言。这种语言称为“面向应用的语言”（application-oriented languages，AOLs，也称为“专用语言”或“liitle 语言”[Bentley1988]）。这是 FAST 方法[Weiss1999]的中心策略，在第 1 章中已经简略地讨论过。

即使已存在的解决方案方法是适合于问题的好结构，它们也有可能不经济或不能满足其他的实际问题。例如，我们可能想要为织染应用中的“颜料混合控制语言”设计一种语言。我们可以将颜料表示成类，使用重载（例如 operator+）来表达颜料的混合。这个问题的语义可以由 AOL 来捕获，但 C++只能在运行时间很好地处理它们。例如，我们可能想要在类型系统中包括颜料属性（比如酸度和碱度）——碱性颜料当然不应当与某种酸性颜料混合。一个精化的类系统可以处理这个问题。但我们来考虑这个例子：尽管 A 和 B 类的颜料可以混合，且 B 和 C 类的颜料可以混合，但 A 和 C 类的颜料绝对不能混合。一个 C++再现很难在编译时找出这种错误：

```
A a;
B b;
C c;
Dye d;
d = a; // mix in initial pigment
d = d + b + c; // whoops, mixing pigments we shouldn't mix
```

如果我们在一个 AOL 中分析这个相同的语句，语言处理器的语义分析将检查出这个错误——它了解 C++运行时语义分析所不了解的领域规则。

AOL 可以比 C++更富于表现力，这对定位于不熟练用户的语言尤其重要。这种表达能力总是与便利程度结合在一起。这就是之所以诸如 **yacc** 和 **lex** 这样的语言能够存在的原因。它们所做的事情都是 C++所不能做的，但它们都以各自的语法（或多或少）简洁地表达了领域语义。

AOL 通常由自动代码生成（如果考虑它，您将获得非同寻常的动力）来驱动。其目标不是生成大量的代码——代码要占用内存进行存储，要占用时间来运行，所以代码意味着更高的代价。AOL 长远的益处是便于编程和易于长期的维护。

但这些好处带来了一定的代价和一些缺点。一个好的 AOL 难以设计，且维护成本较高。时间将揭示改进最初的语言设计的可能性，但让传统程序跟踪它们所编写的编程语言中变化的代价很高。除非语言设计对演进的预见极佳，存在更改语言的压力，但这种更改依然代价高昂。

好的 AOL 拥有分析器、调试器和翻译器的丰富支持环境，可以很容易地跨平台（操作系统、处理器等）重新定位目标。每种环境都必须支持项目当前使用的所有平台，这可以增加技术上的筹码。这种支持成本随着所部署的不同 AOL 的数量而增加。AOL 的好处（便于编程、静态分析、形式验证、调试、自动的文档生成及优化）必须超过开发和维护这种语言本身所付出的代价。

7.3 示例：一个简单的语言翻译器

这个示例将使用过程设计来构建一个递归下降的分析器，用面向对象的设计来处理符号表信息。编译器通常包含多个明确定义的子领域（词法分析、语法分析、语义分析、代码生成及其他领域）的代码。这些领域基本上是互不连接的。每个领域都可以使用单个的主流范型很好地进行管理。

7.3.1 将领域划分成多个子领域

正如我们在第4.2节中所讨论的那样，将应用领域划分成多个互不连接的子领域是很重要的。一个理想的子领域包含可在其他应用中被复用的代码。例如，为编译器所编写的符号表管理代码应当满足连接编辑器、调试器和其他软件生成工具的需求。

选择一种划分

我们不使用正式的过程来将问题解析成多个子领域；相反，我们依靠有经验的设计者的直觉。见识和构建过很多编译器（或者说编译程序）的设计者了解所有编译器所共享的经常性出现的架构的模式。这些领域通常包括以下子领域：

- 词法分析
- 语法分析
- 语义分析
- 符号管理
- 代码生成
- 优化

我们怎么知道这些是好的领域、而这种划分又是一个好的划分呢？理想的划分应当是模块化的，所划分出的（子）领域应当是内聚而去耦的。有些领域自身可以独立地销售，这通常是好的领域的一个充分（但不必要）条件。领域应当尽可能少地交叠，并尽可能少地相互干扰。我们并不想在多个领域中找到相同的共同性。在简单的编译器中，词法分析、语法分析和语义分析是相互独立的“阶段”。词法分析是一个内聚的活动。其代码要做一件事情，并将其做好。在理想的情况下，词法分析器不用关心语法分析问题。我们可以认为词法分析和语法分析已经进行充分去耦，二者都可以作为单独的领域。

在丰富的语言（比如C++）甚至是可移植汇编语言（比如C）的实际编译器中，各个领域并不是完全独立的。一个重要的设计考虑事项就是领域之间（比如词法分析和语法分析之间，或者语法分析和语义分析之间）的相互作用。

即使在这个层次上，划分也可能影响用于实现编译器的编程语言。一门编程语言可能拥有进行词法分析的原语，例如，用 SNOBOL 编写的编译器不可能有单独的词法分析模块。甚至下一个层次的各个领域中还出现了更多的语言灵敏度。例如，基于一个分析程序生成器（比如 yacc）的设计结构与基于一个人工编码、递归下降的分析程序的设计结构大不相同。但两个分析程序与符号表管理和代码生成领域有着相同的关系——除非分析程序生成器或其他实现技术为我们提供了可以属于符号管理和代码生成的差异参数。

领域分析

根据编写编译器的经验，我们将应用分解为多个领域，从而为分析确立了一个起点。现在，我们需要扩展这种分析。正如[Coplien1992]中第 8 章所述，扩展分析提高了复用的可能性。我们可以分两步来扩展这种分析。第一步，查看这些子系统的（明显的）应用和支持编译器的工具族：汇编器、连接编辑器、浏览器等。第二步，扩展到基本上支持语言翻译的工具族，包括其他语言的编译器。

假设领域分析最初是为了创建一个 C 语言支持工具的族。那么此分析是否要扩展到其他语言的支持工具？规范词法分析和语法分析之间的相互影响（这是由诸如 typedef 这样的结构引起的）是很困难的。当进行跨语言的领域分析时，我们在语言的语法分析复杂性中发现了差异性。有的是 LL 语法，有的是 LR（0）语法，有的是 LR（1）语法等。语言的语法分析复杂性影响了语境的灵敏度，因此也就影响了词法分析、语法分析和语义分析的独立性。例如，一个 C 编译器的设计者可能将对 typedef 名字的识别放在词法分析中，以便语法可以像处理任何其他类型一样来处理 typedef。例如，给定语句

```
typedef int Int;
```

紧接着的语言产品

```
type_name decl_list
```

应包含下面两个源码语句：

```
int a, b;
Int c, d;
```

词法分析程序可以通过询问语法分析程序来识别 typedef 名字。然而设计者还有另外一种选择，那就是使用接受下面这种形式的产品的语法：

```
name decl_list
```

假设“name”是类型标识符，此假设在语义分析中必须是有效的。这样的语法减轻了词法分析器识别 typedef 名是否为有效的类型名的负担（但可能为词法分析带来与语

义分析更加紧密的耦合这一不利影响)。架构的这种折中影响了每个领域的共同性分析的细节，也影响了子领域和用来实现它们的模块的独立性。

“粘合”(Glue)领域

有些抽象超出了个体的领域。在一个编译器中，我们将在所分析的所有子领域中发现 string。它究竟属于哪个领域？我们根据常识将 string 放在基本构件块的一个单独领域中。我们通常可以在标准库中、在商业可用的支持包中以及在一般或标准操作系统 API 中找到对这些抽象的支持。正如 GOF 的著作[Gamma1995]中所述，这些有时被称为“工具箱”(toolkit)，它们常用于稍正式一些的分析。

领域分析和迭代

一旦完成了领域分析，设计者就应重新返回到子领域，以确保所使用的的划分标准依然是比较好的划分。领域分析可以通过多种方式（更改领域间的耦合）来扩展抽象，或使其产生斜交。耦合中的这种变动可能暗示了子领域边界的改变。如果是这样，我们就需要根据新获得的认识对问题进行重新划分（通常进行一些微调)。过程迭代一直进行到此过程收敛为止。

例如，假设最初的编译器设计由 **Parser**、**Code Generation**、**Symbo Management** 和其他领域组成。硬件架构应当是 **Code Generation** 领域的一个差异参数，但我们将假设其他领域与硬件无关。设计经验告诉我们：编译器的前端处理器（分析程序）可以利用硬件特性来生成更加紧凑的分析树，这反过来得到了更高效的代码。例如，硬件可能提供了直接调度 case 语句的指令——这将对分析器很有价值。设计者可以在单独的 **Machine Architecture** 领域中捕获这些架构特性从而利用它们。**Parser** 领域和 **Code Generation** 领域都可以将 **Machine Architecture** 领域当作差异参数使用。

7.3.2 在子领域中寻找合适的范型

一旦我们确定了子领域，我们就必须设计和实现每个子领域。如果我们已很好地选定了自己的领域（并可以在可简单划分的系统上取得预期的效果)，处于这个层次的大部分设计都可以在领域之间没有相互干扰的情况下进行下去。

每个领域都必须划分为可管理的部分（比如算法步骤和对象结构)。我们可以将这些部分分组到抽象（比如函数和类）中。这些部分和相关的抽象都应通过隐藏设计秘密（模块性）和将它们的接口与包含它们的子领域的稳定特性结合起来，以支持产品的演进。

对于在某个领域中有经验的实践人员来说，领域中范型的选择通常是很明显的。但对于新的领域，设计者就需要选择合适的范型，以满足软件可维护性的目标。要找到合适的范型，我们要使用第 2 章所介绍的方法来分析子领域的共同性和差异性。这里我们

来研究编译器的 **Sybmol Management** 子领域，并选择一个范型，找到这个子领域的重要抽象。

正如第 6 章中我们所发现的那样，有些范型是 C++所支持的。我们将努力使用这些范型来表达 **Symbol Management** 子领域中的共同性和差异性。

领域词典

对于任何一种编译器，**Symbol Management** 领域都是一个重要部分。我们可以将 **Symbol Management** 当成一个模块，以表明它可以单独地进行设计、开发和配置。如果我们使用领域分析来驱动分析和设计，那么一个设计良好的符号管理模型不仅适合于编译器，还适合于汇编器、连接编辑器、调试器和其他的工具。

领域分析的第一步是捕获领域词典。图 7.2 显示了 **Symbol Management** 领域的词典的最初形式。这些术语是领域中的实践人员所熟悉的。其中很多也是符号表的外部客户（即使用编译器、调试器和连接编辑器的人）所熟悉的。

Typedef：一个 `typedef` 子句，是另一种类型的别名

标识符（Identifier）：自定义类型、函数、“变量”或标签的任意标识符

函数名（Function name）：独立或成员函数的完全合格的名字

行号（Line number）：编辑单元中的源代码行号

标签（Label）：配合 `goto` 的使用

大小（Size）：以字节表示的数据元素的大小

结构标记名（Structure tag name）：……

地址（Address）：……

偏移量（Offset）：……

存储类（Storage class）：……

作用域（Scope）：……

结合（Alignment）：……

图 7.2 **Symbol Management** 词汇表

领域的共同性分析

现在我们已经建立了一个领域词典，接下来要开始在领域中寻找结构。有些词典条目自然地与其他条目分组在一起。这些分组中的每个条目都形成了一个抽象族。族成员共享共同的属性，并通过差异性相区别。如果编码这些共同性和差异性，我们就可以将它们与直接表达这些设计维度的编程语言的特性结合起来。我们将分析捕获在一个差异性表中。结果将是适合于每个领域的范型或范型清单。

注意，不要直接从领域词典过渡到差异性表。要尽可能用直觉来形成领域的抽象。我们可以使用经验规则和通过建议可能的地方来寻找拥有广泛基础的抽象，并以此指引

这种直觉。Weiss[Weiss1999]建议按照下列顺序从三个方面来审视设计：

1）领域的外部接口中的抽象。

2）领域中的工作单位（例如，内部状态和状态转换）。

3）其他内容。

符号领域很简单，其分组是直觉性的。词汇表条目 **Typedef**、**Identifier**、**Function Name**、**Structure Tag Name** 和 **Label** 都是各种各样的“名字”。它们建立了差异参数的值的范围，所以在共同性分析表中，它们中的每一个都可能作为单个的行出现。子领域 **Scope**、**Alignment**、**Size**、**Offset**、**Line Number**、**Storage Class** 和 **Address** 都是符号表条目的特性。我们可以将一个 **Line Number** 当作一种特殊的（隐式）标签，它描述了代码中的一个位置，我们可以暂时将其当作一种退化的 **Name**。

我们根据直觉建立一个共同性领域 **Name**。**Name** 是跨领域词典中的 **Typedef**、**Identifier**、**Function Name**、**Structure Tag Name** 和 **Label** 这些条目的抽象。所有的 **Name** 在很多方面都是相同的。我们要捕获和描述这种共同性来驱动共同性分析。所有的 **Name** 共享什么样的特性？它们的行为类似，我们可以将它们存储在符号表中，将他们组织到搜索表中，寻找它们的名字字符串等。我们找到了“行为”的（behavior）共同性。根据业务直觉，我们从领域的已发布的（外部）接口中的抽象来驱动主要的共同性。

经验和直觉可能告诉我们：**Line Number** 和 **Label** 不属于 **Name** 子领域。**Line Number** 不是一个 **Name**。我们可以考虑一个更宽的领域（可能是 **Symbol**）让它同时包含 **Line Number** 和前面我们已归入 **Name** 子领域中的其他符号表特性。分析到这一步时，我们就可以遵从这种直觉，抽象出一个更高的层次。如果我们已将所有这些词汇条目当作 **Symbol**，我们就错过了将 **Name** 捆绑在一起的一个重要的共同性轴。所以仍然要将 **Name** 当作一个子领域，这样我们将回过头来协调 **Name**，将它与 **Line Number** 和 **Label** 一起归为更抽象的 **Symbol** 领域。或者，我们可以将 **Line Number** 和 **Label** 当作消极差异性来处理。

我们还知道，大部分（尽管不是全部）的 **Name** 都拥有一些形式的地址或偏移量以及指定的作用域（**Name** 可以在这些作用域中找到）。这些都是所有 **Name** 共同的结构元素或数据字段。至此，我们找到了“结构”（structure）的共同性。这符合 Weiss 关于寻找共同性的第二个方面的建议：内部状态。可以将内部结构的另一个小的方面（用于构建 **Name** 结构的磁盘表示法的算法）当作共同的内部结构或共同的外部行为。二者都建议围绕着 **Name** 抽象的相同分组。

方案领域是否提供了表达这些共同性的机制？查看表 6.2 的“共同性”那一列，我们就会找到“相关的运算和一些结构”这个条目。C++可以捕获和表达这种共同性，所以我们可以将 C++作为方案领域。

现在，我们已经理解 **Name** 是怎样相似的，那么它们又是怎样区别的？尽管所有的 **Name** 都使用相同的基本程序定制磁盘映像的符号表条目，我们还是发现了这样一个算法族，它的成员与符号表条目的类型不同。

表 7.1 所示的共同性表中捕获了不同性或差异性。主要的差异性是符号的种类或类型：typedef、标识符、函数名、标签、行号或 Struct（结构）标记名。我们在运行时间将类型捆绑到一个给定的族成员；这就允许我们更抽象地推理使用符号抽象的应用代码中的类型。再次查看语言共同性表（表 6.2）中“相关的运算和一些结构”这一行，我们看到，对于算法、数据结构和状态中的一种差异性和运行时绑定，都应当使用 C++虚函数。我们将对此设计使用的主要范型是对象范型。这种映射已经以斜体注释记录在表 7.1 **Symbol Type**（符号类型）行的最后一列中。（表中的斜体条目是在转变分析期间添加到表中的注释。）

表 7.1 对共同性领域 Name（共同性：结构和行为）的编译器转变分析

差异参数	意 义	领 域	绑 定	默认值
Symboi Value *对象值*	函数、标识符等的实际名字	[a-z A-Z] [a-z A-Z 0-9]	运行时间	无 *状态值*
Symbol type *结构* *算法*	主要类别的符号的使用和意义	typedef、标识符、函数名、标签、行号、struct 标记名	运行时见	无 *虚函数*
Scope *数据*	符号所出现的范围	全局、文件、函数、类	运行时间	无 *枚举（enum）*
Layout *算法（因为算法产生布局）*	在符号表文件中，不同符号有不同的状态数，却有惟一的布局	typedef、标识符、函数名、标签、行号、struct 标记名	编译时间	0 *数据值*
Alignment,size, offset,address *数据*	符号条目的数据值特性	不同约束的整数	运行时间	0 *数据值*
Storage class *数据*	对数据条目是如何存储在应用程序中的描述	register、static、auto、extern	运行时间	0 *枚举（enum）*
Symbol table formatting *算法*	在对象文件中，每个符号都有自己的外观	不规则	运行时间	无 *虚函数*

以同样的方法浏览表 7.1 的其余行。我们用“枚举”（enum）处理 Scope（作用域），用继承来处理 layout（布局），用简单的数据值差异来处理 alignment（结合），用“枚举”来处理 Storage Class（存储类）。因为我们选择虚函数来支持前面的类型差异性，所以已经

包含了继承。支持布局中的差异性的继承与支持表达不同符号类型的虚函数差异性的继承恰好在同一条线上，所以我们不需要对“布局”进行别的处理。数据值差异的处理总是很琐碎的。注意，设计包括我们从图 6.2 中所选择的行中的数据结构和状态。枚举的行为很像数据值：它们只是在我们用来组织成继承层次结构的类中“凑凑热闹”而已。

注意，表中有两行几乎相同。第二行捕获符号类型的差异性：typedef、标识符、函数等。最后一行捕获了格式编排算法中的差异性。这两行都定义了一个单独的差异性范围。但是，这两个差异参数导致了设计中同种类的变化：运行时的结构和格式编排。我们说这两个差异参数是协变的（或者说相关变化）——即格式编排算法与符号类型有密切的关系。因为这两行都建议使用同种范型（虚函数）作为解决方案，所以我们可以认为其中的一行是多余的（第二行[①]或者是最后一行，但并不是两者）。如果这两行交叠在一起，那么项目文档就应同时捕获两个参数的描述。

我们使用图 7.3 所示的差异性依赖（关系）图来形象地描述这种并项。图中的节点（椭圆框）是子领域。中央的节点代表当前分析的焦点，即 **Name** 领域。周围的节点代表差异参数，分别标以参数名。后面（第 8.4 节）我们将差异参数本身当作子领域，图中的虚箭头捕获了领域依赖性。这里，差异参数确实对应于领域，但我们用这种表示法暂时只是为了捕获领域及其差异参数之间的关系。这里略去了可以表达为数据值中的不同的差异性，因为它们并没有影响抽象的结构。

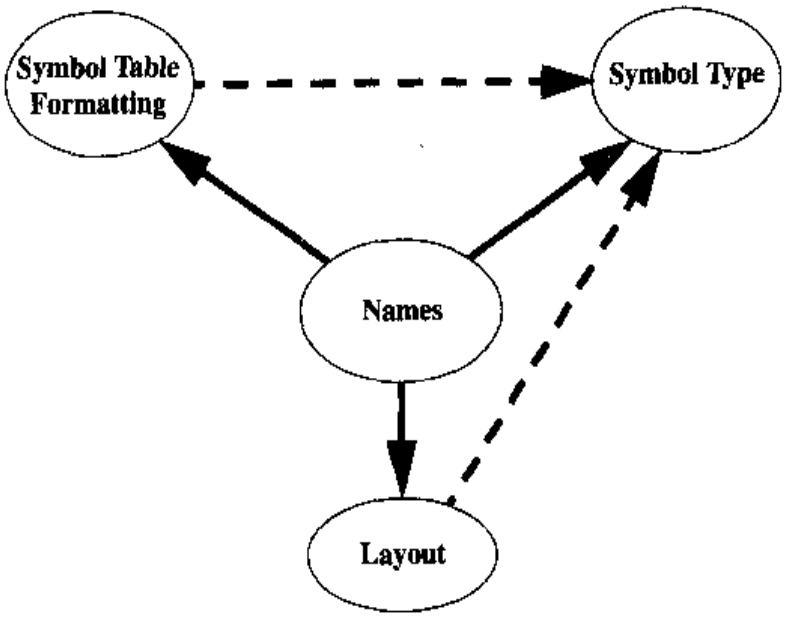


图 7.3 共同性领域 **Name** 的差异性依赖（关系）图

① 原文为 first。——编者注

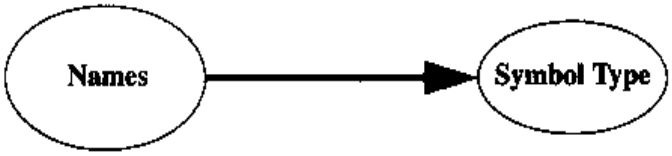


图 7.4 简化的共同性领域 NAME 的差异性依赖（关系）图

除差异性表中的显式依赖性以外，表中的行与行“之间”还存在其他的依赖性。图 7.3 包括了 **Layout** 和 **Symbol Type** 之间、**Symbol Table Formatting** 和 **Symbol Type** 之间的虚箭头，以显示这些领域之间的依赖性。实际上，**Symbol Table Formatting** 只依赖于 **Symbol Type**。差异参数 **Symbol Table Formatting** 和 **Layout** 是相关变化的。当两个参数相关变化时，可以删除子图中包含这两个参数的非叶节点。**Layout** 只是将 **Symbol Type** 的影响传递给 **Name** 领域，**Layout** 自身对 **Name** 没有其他任何影响。如果对图 7.3 中的所有差异参数进行上述处理，结果依赖（关系）图就只剩下两个节点：**Name** 和 **Symbol Type**，如图 7.4 所示。如果将此图中的每个节点当成一个领域，那么就可以将 **Layout** 当作 **Symbol Type** 领域的一部分。对 **Layout** 重要的就是对 **Symbol Type** 重要的。

通过使用继承和虚函数，我们已经选择了使用对象范型。我们用迂回的方式获得了这个决定，或许很多读者在看这些分析之前就已经得出这个结论了。进行分析的原因是问题并不总是很简单，踏踏实实地找出适当的对象范型总比使用不当的范型要好。

仅有共同性分析是不够的。对象范型还有很多其他的设计规则，说明如何高效地使用继承。我们应当按类型将符号表的条目组织到特化（专门）的层次结构中，并将这些层次结构直接放到继承树中。对符号抽象之间的特化关系的分析本质上不属于共同性分析，但它是对象范型所特有的。共同性和差异性分析引导我们得到带有正确的差异参数的正确范型。对精明的 C++设计者来说，其余的问题都是“SMOP”（简单的编程问题）了。

回到Line Number（行号）和Label（标签）的问题

前面我们注意到：**Line Number** 和 **Label** 并不适合于共同性分析。现在是回过头来看看这个问题的时候了。这里我们将讨论两种选择。第一个选择是将 **Line Number** 当作一种消极差异性。第二个选择是用一个子领域的层次结构来解决这个问题。子领域层次结构的解决方案更加通用，它通常会比消极差异性的方法带来更具可扩展性的设计。

消极差异性（第 3.3.2 小节）是假设“共同性分析得到了某种共同性，除了……”的一种方法。我们可以假设一个 **Label** 或 **Line Number** 是一个 **Name**，但它与其他名字没有共同的类型特性。我们可以忽略多范型设计中归到 **Label** 中的类型特性，并期望没有人试图实际地去解释它。也可以在语言层次“关掉”类型特性，以便设计者不会偶尔还

要依赖它们。怎样才能做到这一点？

我们使用公有继承来表达符号领域中的共同性。为表达消极差异性，必须取消一些从基类到派生类的属性。这称为“带取消的继承”。在C++中带取消的公有继承是非法的，因为它违背了类的派生和子类型的导出之间的并行性，而这种并行性是C++所强制要求的。应用带取消的继承是面向对象的编程语言中最公认的消极差异性的形式之一，有关此问题的详细讨论，请参阅[Coplien1992]中的第6章。我们可以使用第6.11节所述的一种其他方法，但要在可能的时候避免一个消极差异性。

从一个不同的观点来看，很多消极差异性可以看作积极差异性。前面我们已经暗示了这种解决方案。可以寻找一种更抽象的共同基础，而不是将**Line Number**和**Label**看作退化的**Name**（退化表示它们没有类型信息）。是否存在这样一个更高的领域：**Name**和**Line Number**都是此领域的子领域？回答是肯定的，直觉上我们将这样一个领域称为**Symbol**。现在我们拥有了一个领域的层次结构，顶点（顶端）是**Symbol**，下面是**Name**、**Line Number**和**Label**，**Name**领先于其下一级的领域。

因为我们正在使用对象范型，所以这些层次结构是继承层次结构。注意这样一点是很有益处的。最后我们确定的顶层领域名为**Symbol**，这正是第7.3.2小节开头部分我们所选择的领域。在实现中我们将看到这个层次结构。

7.3.3 实现设计

在此处，设计者通常可以直接捕获代码中的结构。如果设计者觉得使用图像表示法可以帮助阐明设计，或者图片可以让开发组织的其他人员更有效、更方便地认识设计，那么设计者就可以使用一个临时的图像表示法。图7.5所示的框架化代码是从设计中直接抽出来的。这个继承层次结构反映了我们从对象范型获知的另一种共同性：基类捕获了全部派生类的共同行为和结构。构建继承层次结构变成了将类按它们的共同性分类的简单工作。

共同性分析已经“揭示”：对象范型是适合于符号领域的范型。感觉就好比多范型方法已经完成了它们的任务，此时，对象范型可以“接管”过来了。现在是离开Booch和Rumbaugh，开始使用对继承层次结构非常有用的子类型导出和继承规则的时候了。

如果方案共同性表中没有与我们对符号管理的分析中所发现的共同性相匹配的条目时该怎么办？设计者可能想将此问题强制性放入到一个C++可以表达的共同性维度中。没有经验的设计者对每个共同性都使用继承，这是应当避免的，因为这样做扭曲了结果系统的架构，使得系统难以演进。我们需要继承以外的范型，通常，我们还需要一些C++无法表达的范型。第7.6节中我们一定程度地讲述了这个主题，并在第8章进行了更广泛

的论述。

```
class Symbol { . . . . };

    class Name: public Symbol {
    };

        class Identifier: public Name {
        };

            class FunctionName: public Identifier {
            };

        class Typedef: public Name {
        // not an identifier!
        };

            class StructTagName: public Typedef {
            // in C++, a structure tag is a typedef
            };

    class LineNumber: public Symbol {
    };

    class Label: public Symbol {
    };
```

图 7.5 捕获表 7.1 中分析的某些 C++类接口

7.4 设计，而不再是分析

现在我们回过头来看看已经完成了哪些工作。我们运用了自己的领域知识以争取获得最大限度的便利。最粗略的框架区分源于我们的直觉，并需要有编写编译器的经验作为支持。（如果您没有编写过编译器，请仔细考虑我的话。）其他架构可能同样适用，大部分问题都有多种可行的解决方案。

7.4.1 分析、架构或设计？

多范型设计活动最适合分析、架构或设计的传统意义吗？多范型设计活动并不是经典意义上的分析，因为我们是组织而不是捕获领域知识。我们已经考虑过结构和划分标准，并且聚焦在哪些是“如何”（就结构的意义而言），它超出了“什么”（就行为的意义而言）。这里用来组织领域知识的方法也可以在我们获得新的领域知识时用来组织新的领

域知识。我们可以将词汇表条目群集到相关的子集中，这给予了我们关于领域结构的很有价值的线索。但最终的划分仍然要利用我们的见识和直觉。最好的情况是，划分也利用我们对系统如何演进的预见：好的架构是封装了变化的架构。这里，经验和直觉再次成为我们的最佳的向导——历史通常是未来最好的预言者。

如果架构的第一层是根据应用领域标准得到的一个粗略划分，那么架构的第二层就创建了方案领域结构（比如 C++编程语言所支持的这些结构）所支持的抽象。这看起来像是突然的转换，但我们认为这是必要的，并且是比最开始出现时更加直接的转变。C++允许我们构建自己的抽象。通过在抽象上构建抽象，我们常可以将可表达性的层次提高到与领域词典相同的层面。从这种意义上来说，C++是捕获和构建领域词典的恰当工具（尽管它可能是最缺少理性层次的表达）。

我们来考虑一下如今大名鼎鼎的 Whorf 的话："语言形成了我们思考和确定我们可以表达什么内容的方式。" [Whorf1986] 如果实现语言是 Smalltalk，那么在架构的第二层上我们会选择不同的抽象（第一层是直觉性的领域划分）。如果实现语言为 CLOS，我们仍然会选择不同的架构。如果实现语言是 APL，那我们就会选择别的内容。这就是为什么围绕着 C++编程语言的形式和基础形成架构很重要的原因。围绕 C++结构而形成架构并不意味着 C++是"最佳"或"正确"的语言。我们可以从共同性和差异性表中为 Smalltalk 得出一个不同的架构。

尽管多范型设计依据 C++语言所支持的内容来形成架构，但这并不意味着它是低级的或后端的方法。语言的考虑事项和所关心的问题涉及到最初的设计考虑事项。早期的设计对编程语言的考虑要少于后端的设计和实现。但这并不意味着高层次抽象中就不涉及编程语言。

7.5 另一个例子：自动微分

Max Jerrell 介绍了一个能够提供自动微分数学函数的简单而强大的设计 [Jerrell1989]。本节我们就来研究一下自动微分领域，并为这个应用得出一个设计。这个应用演示了对非面向对象的编程的设计结构的有效使用。也说明了分离应用领域和方案领域的困难。

长时间以来，计算机就被用于计算和打印跨范围的独立变量的数学函数的导数值。反过来，这些导数又可以为其他相关领域所用，比如寻找原函数的局部最小值或最大值（因为这些地方的导数为 0）。所以现在假设我们试图解决这样一个问题：

用尽可能简便的设计和编程来计算复合函数的导数。

计算导数有三种经典的方法：

1. 直接计算导数作为给定点的函数斜率，用 y 方向的变化率除以 x 方向的变化率：

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

程序员需要提前得出导数的公式。计算机完成所有必要的工作，任意函数 f 的导数可以用封装在函数 fprime 中的算法获得，可以使用牛顿法或者直接求商法：

```
const double h = 0.001;

double differentiate(double f(double), double x) {
    return (f(x + h) - f(x)) / h;
}
```

但精度与 h 的值紧密相关。这种方法可能不适用于（即使用了迭代方法）斜率很大的函数。

2. 让程序员手动对函数（比如 f）进行微分，并将导数编码为一个新的函数 fprime。例如，要获得这样的代码：

```
double f(double x) {
    double a = sin(x);
    return a * a;
}
```

程序员需要编写下面这个差分函数：

```
double fprime(double x) {
    return 2 * cos(x) * sin(x);     // derivative of
                                    // sin squared
}
```

但这就需要程序员完成大部分的工作（计算出导数），并且可能很慢、容易出错，尤其是对于很长的方程式。

3. 第三种选择是使用“自动微分”（automatic differentiation）。自动微分要求程序员指定简单的导数。例如，程序员必须告诉计算机 sin 的导数是 cos。但也可以“教会”计算机关于导数的加减乘除基本运算的微积分的规则。例如，和的导数为导数的和。我们可以使用对象范型的数据抽象来处理函数的复合。例如，要表达：

```
g(f(x))
```

可以让计算机计算 f(x) 的导数信息，然后将结果作为对象进行传递（包含了导数信息）作为函数 g 的参数。导数信息可以作为梯度矢量（一阶导数）和 Hessian 矩阵（二阶

导数）来传递，计算其中内容的公式由微积分公式规则来指定。所有函数的值（包括常值函数和简单参数）都可以当作统一的 Value 类型来处理。（这种肤浅的描述省去了很多重要的细节，但这是经过了检验的方法，在书后的相关文献中获得了证实，请参阅[Jerrell1989]。）

我们很容易就能注意到：上述每种方法都会带来不同的设计，也就是针对同一问题的不同解决方案。这个示例说明：在单独的问题结构中，设计结构可能不是很明显。当我们了解了实现策略的某些内容（比如确切的方案技术）时，方案结构也就显现出来了。

将这些实现因素捕获为问题陈述的一部分通常是很方便的。原来问题的定义映射为计算导数的一般术语。但我们可以选用下面这种描述：

用自动微分法来计算复合函数的导数。

这看起来是让问题的陈述涉及到了有关的解决方案事项。所以我们想要推迟考虑解决方案事项，因为绑定它们会过早地限制设计选项。但如果设计选项完全改变了问题的结构，那么就应当将它们作为问题的一部分。最好作为必要条件的这种方案选择有很多：集中式与分散式处理之间的选择，实现语言之间的选择，操作系统或计划进度之间的选择，以及其他选择。不可轻视这些问题。

我们要对自动微分程序进行软件设计：找到领域、建立它们的差异参数、选择适当的范型并编写代码。

要找到问题领域，我们可以考虑图 7.6 中所示的问题定义中的族分组。基本运算也形成了一个族，这个族主要的共同性是它们的导数可以不用费力地机械地求出。其他函数（比如 f(x)=sin(x)+cos(x)）是否形成一个族？在自动微分法中，我们将它们当作更大的族 **Value** 的一部分。**Value** 的所有成员共享相同的结构——捕获了它们的状态的矩阵和值，它们还共享任何代数表达式所相同的外部行为。最后，有一个差异参数 **Degree**，它可以导出其他两个族的结构。我们将这个差异参数自身看作是一个子领域，与第 8.4 节的论证相符。

Value（值）：代表一个表达式、表达式的值及其在特定纵坐标值处的偏导数的抽象

Basic operation（基本操作）：*、-、+、/（加减乘除）操作

Degree（次数）：不同次（比如，矩阵变化的形式）的问题具有不同结构的解决方案

图 7.6 自动微分领域词汇表

7.5.1 基本运算领域

表 7.2 显示了对子领域 **Basic Operation**（基本运算）的分析。这些分析是对两个参数（可能是函数、标量、常数或其他的 **Value**）进行运算的“函数”（加减乘除），微积分学为它们提供了正式的求导规则。因为这些函数非常通用，所以它们的导数可以进行规范化处理。又因为它们是内建的 C++运算，所以我们可以将它们归为一个共同的、可表达的运算的族。

表 7.2 对 Basic Operation 领域的转变分析（共同性包括数量、参数类型和返回类型）

差异参数	含 义	领 域	绑 定	默认值
运算类型 *算法*	生成导数（及其值）的算法随着运算类型而不同	*、-、+、/	编译时间	无 *重载*
等式的次数 *结构* *算法*	独立变量的个数	正整数	编译时间	1 *全局* *常量*

7.5.2 次数领域

一个代数表达式包含一个或多个独立的变量。表达式的“次数”（degree）是指独立变量的个数。我们可以求关于这些变量中的任何一个或者所有这些变量的微分。每个导数都是关于对应变量的导数。**Basic Operation** 领域的每个成员都必须知道自己所要处理的系统的次数，因为它必须求其关于每个独立变量的参数的微分。这样 **Degree** 就成为 **Basic Operation** 领域的差异参数。

7.5.3 值领域

自动微分计算进行的方式与任何其他代数计算相同。也就是说，最内层的子表达式先计算，计算结果由表达式树中高一级的运算进行组合。在普通代数运算中，每个子表达式的结果是一个标量值。在自动微分法中，运算的结果不仅包括标量值，还包括表达式的一阶导数（梯度矩阵）和二阶导数（Hessian 矩阵）的系数。这就表明表达式是否为一个常量、标量、一种基本的运算或任意自定义的函数。这些抽象形成了一个族或领域，同样，我们称之为 **Value**。

表 7.3 显示了子领域 **Value** 的转变分析。注意，Hessian 矩阵和梯度矩阵的使用是这个子领域的一个显式共同性。这有一点奇怪。如果回过头来考虑其他的替代解决方案，

比如牛顿法或手动微分，那么这些矩阵则是相对于微分技术领域的更宽背景的差异性。这是一个作用域的问题。这里涉及到自动微分领域内的抽象族，而不是通常意义上的微分领域。自动微分依赖于这些矩阵，因此，**Value** 也就成为我们所关注的一个领域。

如果我们没有选择自动微分，高层次的领域分析自身将有所不同。使用牛顿法时的子领域与使用自动微分法时的子领域是不同的，符号微分法将由另一种领域结构来支持等。解决方案技术的选择可以改变领域分析，这是迭代开发的一个有力论点。这在新的领域中尤为重要。

Hessian 矩阵和梯度矩阵是结构共同性的一种形式，因此建议将继承作为一种实现技术。但是，使用继承与基类和派生类的结构完全无关——该继承只用于实现的共同性。在这种情况下，机敏的 C++程序员可能倾向于使用私有继承或 **HAS-A** 关系。但我们用公有继承来定义 **Value** 领域的族成员，总之，我们的目的是捕获算法中的差异（表 7.3）。

表 7.3 Value 领域的转变分析（共同性：包括 Hessian 和梯度矩阵的使用、当前值的表示方法以及大部分的行为）

差异参数	意　义	领　域	绑　定	默认值
值与导数的计算 *算法*	每种类型的值都有计算其值、其矩阵值及系数等的算法	常量、标量、所有类型的函数	编译时间	无 *继承*
等式的次数 *结构* *算法*	独立变量的个数	正整数	编译时间	1 *全局* *常量*

图 7.7 显示了前面所讲述的这些领域的差异性依赖关系图。请注意 **Degree** “领域”。我们没有将 **Degree** 看作一个真正的领域，它只是定义了所研究的等式的次数。但它是 **Value** 和 **Basic Operation** 领域的差异参数，也是设计的一个重要方面。

图 7.7 自动微分法的差异性依赖关系图

Degree 领域是一个简单的数值型参数，我们可以将其表示成一个全局 C++ `const int`，此类型可以很好地适用于等式的单一系统。如果要解决的是等式的多个系统，我们可以将每个系统的实现的作用域限制在其自身的类中，这些类拥有各自关于等式的次数的值，

记做符号 NDegree：

```
const int NDegree = 3;      // suitably scoped
```

Value 领域带有这种值和导数矩阵。因为它通常包含了相关的数据项，所以我们将其设计成一个类。该类的声明如下：

```
class Value {
friend Value operator+(const Value&, const Value&);
friend Value operator-(const Value&, const Value&);
friend Value operator*(const Value&, const Value&);
friend Value operator/(const Value&, const Value&);
public:
    Value():
        gradient(NDegree),
        hessian(NDegree, NDegree) { }
    operator double() const { return value; }
protected:
    double value;
    vector<double> gradient;
    matrix<double> hessian;
};
```

使用 friend（友元）关系只是为了保证对一般运算的访问权限。

Basic Operation 使用直接的重载，将操作符自身（+、-、*、/）当作差异参数。我们手动编写各运算符的算法的编码，编译器会在编译时间根据语境来选择相应的算法。基本运算的代码如下：

```
Value operator+(const Value &x, const Value &y) {
    Value f;
    f.value = x.value + y;      // operator double()
                                // eliminates
                                //   the need to say y.value
    for (int i = 0; i < NDegree; i++) {
        f.gradient[i] = x.gradient[i] + y.gradient[i];
        for (int j = 0; j < NDegree; j++) {
            f.hessian[i][j] = x.hessian[i][j] +
                                        y.hessian[i][j];
        }
    }
    return f;
}
```

每个族成员不仅要计算返回值，还必须计算其相关的梯度矩阵和 Hessian 矩阵。它根

据标量值及其操作数的一阶导数和二阶导数来计算这些结果。下面再给出一些示例：

```
Value operator*(const Value &x, const Value &y) {
    Value f;
    f.value = x.value * y;
    for (int i = 0; i < NDegree; i++) {
        f.gradient[i] = x.gradient[i] * y + x *
            y.gradient[i];
        for (int j = 0; j < NDegree; j++) {
            f.hessian[i][j] = x.hessian[i][j] * y
                + x.gradient[i] * y.gradient[j]
                + x.gradient[j] * y.gradient[i]
                + x * y.hessian[i][j];
        }
    }
    return f;
}

Value operator/(const Value &x, const Value &y) { . . . . }

Value operator-(const Value &x, const Value &y) { . . . . }
```

具体的原函数可以在应用需要时进行添加。它们也可以将函数（cos、sin 等）当作差异参数，在编译时间由编译器根据语境来选择。参见下面的示例：

```
Value cos(Value &x) {
    Value f;
    f.value = cos((double)x);
    double fu = -sin((double)x);
    double fuu = -f;
    for (int i = 0; i < NDegree; i++) {
        f.gradient[i] = fu * x.gradient[i];
        for (int j = 0; j < NDegree; j++) {
            f.hessian[i][j] =
                (fu * x.gradient[i] * x.gradient[j]) +
                (fuu * x.hessian[i][j]);
        }
    }
}
```

更复杂的函数可以写成它们的自然代数形式，例如：

```
Value w = exp(t) * cos(omega * t);
```

这种形式使用了恰当定义的 exp 和 cos（如前所述）函数，并两度使用了重载操作

符*。框架开发者必须编写基本运算（包括四则运算函数及其他基本函数）的代码。完成这些工作后，框架使用者就可以编写表达式（比如前面所述的普通 C 表达式），此时该表达式具有对自身进行微分的“智能”。

参考书目中的[Jerrell1989]显示了如何使用这样的设计来优化函数，并解释此设计是怎样支持其他应用的。

此设计不是面向对象的，多范型设计帮助识别相应解决方案技术（比如捕获问题的重要抽象的重载）。这些“问题”抽象中有很多预见了解决方案的结构。也就是说，找到适合于自动微分、牛顿法和手动方法的单个领域结构将会比较困难。

7.5.4 演进设计

领域分析的最重要目标就是支持复用和减少演进成本。好的设计对演进是进行了预期的，好的结构可以很轻松地适应变化。在一个好的结构中，差异参数表达了哪些是可能变化的内容。

本设计中存在两个层次的复用。很多用户复用此框架来计算任意代数表达式的标量值及导数。可以在不改变前一页的任何框架代码的情况下写出新的表达式并计算其值和导数。我们可能想要更改框架自身，将新的微积分计算规则加入到其中，这些改变也应当很方便地进行。

本设计中有多个子领域拥有差异参数。设计预期了（基本运算符）运算类型、（非内建运算和函数）算法、次数中的变化。我们应当可以通过一个全局参数轻易地更改等式的系统次数，并且应当可以更改算法（本例中是添加算法）以计算新的函数的值和导数（例如需要三角正切而添加 tan），甚至还应当可以添加一个新的原函数及其计算和求导的定义。

假设我们要定义内建的 operator^ 的语义来表示取幂（当然，不能添加 C++程序无法理解的运算符，而且，对于任何内建的运算符，都必须遵循 C++规定的这些运算符的优先次序和结合规则）。这是一种相当直接的更改——我们只需要向框架中添加函数：

```
Value operator^(const Value &x, const Value &y) {
    Value f;
    f.value = pow(x.value, y);
    for (int i = 0; i < NDegree; i++) {
        f.gradient[i] = y * x.gradient[i];
        for (int j = 0; j < NDegree; j++) {
            f.hessian[i][j] = x.hessian[i][j] * y
                + x.gradient[i] * y.gradient[j];
        }
```

```
        }
        return f;
    }
```

7.6 外部范型

期望用一种编程语言来捕获所有可能的分析抽象是不可能的。通用语言（比如 C++）很宽泛但却不够深入。有些用 C++无法支持的问题用这些专用的、易于理解的范型就可以很好地获得完满的解决。数据库和分析程序生成器就是可捕获实现技术中的重要设计结构的工具的例子。明智的设计者会避免强制使用 C++特性来表示此架构结构，而是对于具体问题运用适当的工具。

我们可以为设计者的工具箱中每个这样的工具创建一个共同性和差异性分析。多范型分析可以方便地进行扩展，从而在应用领域分析和可用的方案领域之间获得最佳的适用性。这里将此种方法作为练习留给读者——或者留给寻求论文主题的博士。在现实中，这样详尽的分析将使人精疲力竭。正因为我们依赖经验和直觉来将问题分成多个切合实际的领域，所以也要依赖经验和直觉来为这些领域选择合适的范型。

现在回过头来看第 7.3 节编译器的示例。它的领域之一是语法分析。我们对语法分析（产品、动作、缩减、目标等）的词汇表进行共同性分析，然后在 C++方案领域分析中寻找匹配的共同性和差异性的模式。如果比较幸运，我们无法找到一种匹配的模式。如果不幸地强制让分析与之匹配，那么我们将“找到”使用 C++所支持的设计模式来设计语法分析器的方法。这样就错失了使用类似 **yacc** 和 **bison** 这些工具的机会，而它们可能是非常适合于此领域的问题、共同性和差异性的。

因此，通常的设计策略是根据适合于问题的共同性和技术，将领域分成多个子领域。仅用 C++往往是不够的，我们要使用一些“明显的”技术（比如分析程序生成器、数据库、已存在的 GUI 框架和状态机）对其余的子领域进行多范型分析。

7.7 管理问题

范型和技术比管理策略对项目成功与否的影响要小。本书当然不能同等篇幅地介绍管理实践，要了解更加详细的内容，读者可以参阅 Goldberg 和 Rubin 最近的著作[GoldbergRubin1995]。其他比较突出的参考书还有[McConnell1997]和[Cockburn1998]。有些多范型设计活动和判定点让项目管理变得清晰，本节我们将稍涉及一些这方面的问题。相关的主题请参见第 2.2.2 小节“领域词典团队”，第 2.5 节“回顾共同性分析”，第 4.1.4

小节“领域分析活动”，以及第 4.3.2 小节“子领域分析活动”。

7.7.1 Occam 的 Razor：让事情保持简单

使用大多数范型都是没有效果的。范型可以帮助定义一种贯穿其词汇表和世界观的文化。共享的文化是一个成功开发团队的非常重要的一部分。开发文化可以只支持很少的一些范型和工具。另一方面，对相应的工作运用相应的工具也很重要，这样就可以避免由单一范型造成的思想僵化。成功的项目需要对这两方面进行平衡。

应当尽可能地进行测试，因为各种组织很少能避免依赖于各个工具版本的特性。这有时使得在不重新编写其他工具的领域中的代码的情况下升级某个工具的版本变得困难。如果工具的数量不多，那么项目在前向进行过程中可容纳工具集的变化的机会就会增加。如果工具的数量众多，或者工具引入了领域间的耦合（因此就涉及到其他的工具），那么让项目跟踪技术的进步就比较困难了。

组织工作经验为我们提出了这样的经验规则：组织几乎无法同时管理超过三个以上的“主要变化”。“主要变化”可以是新的设计范型、新的管理范型、新的硬件平台或操作系统、新的语言或与客户之间的新的工作方式。项目一旦建立起来，它就可以容纳渐进的变化。通常，首先出现在项目中的是形成长期系统架构的技术选择，我们必须一起考虑它，并将其引入到项目中。显然，范型的数量应当保持得少一些。随着时间的推进和系统市场的扩展，尤其是当项目可以定位它们的影响时（对单个子领域、组织、处理程序或本地环境的其他一些领域），项目可以试探使用新的工具、技术和范型。应当逐渐引入新的范型，并要依靠一个相对稳固的基础。另一条好的经验规则是：在第三个发布版本以前不要引入新的范型。我们应当记住将同时变换的数量限制为三个这条经验规则——数量越少越好。

这里介绍的多范型设计的一个主要功能是它试图用单个通用实现平台（C++编程语言）作为一堆范型的交付工具。这就大大减少了引入 C++可以表达的这些范型（对象、抽象数据类型、过程和类属过程、参数函数和模板等）所带来的冲击。在考虑“三种变化”的经验规则时，我们总想将所有这些范型当成单个的范型，但这对 C++不支持的范型（Java 中的多线程执行、Prolog 的基于规则的抽象、或 ML 的功能性抽象）来说也无济于事。有些技术可以在 C++中模仿这些形式（参见[Coplien1992]的后几章）。如果编程人员非常精通 C++，则可以使用 C++结构容纳这些范型，就当作没有“三种变化”的限制。但这样的组织非常少，大概只有百分之一。（最好不要认为自己会隶属于这样的组织，人们总有得出错位结论的倾向。）

保持较少的工具总量可以减少（工具购置或许可）成本，也可以减少培训时间（和

成本）及其他支持设施的费用。

项目一旦分成了多个子领域（参见下一节），Occam 的 Razor 就可获知哪些范型或工具在选择过程中进行了折中。例如，一个系统可以有五个领域，其中有四个适合在 Java（可能是为了能并行操作）中实现，另一个适合在 C++中实现（可能是为了效率）。除非有其他的情况，否则最好减少其工具总量，调整最后一个子领域，让其适用于一个 Java 实现。

7.7.2 分而治之

对于复杂的问题，大部分人在文化上就习惯于运用拆分的思想。可维护架构的尺度之一就是可以作为独立部件的集合来看待的程度。这种架构可以由更小规模的团队来维护，如果在整体结构下组织相同的代码，这种架构可以更加独立。

经验、对市场的了解和直觉通常是一种好的领域划分的最佳基础。我们应试着将问题划分成解决方案可独立交付的子问题。

系统一旦进行了子划分，设计者就可以根据每个子领域的结构评估其适合何种范型。有可能（甚至很普遍）所有的子领域都使用同种范型。这将是很好地适用于 Occam 的 Razor 的偶然结果。但一个项目首先要明确描述其子领域的划分，然后为每个子领域选择适当的范型，然后再选择支持这些范型的工具。此时，进入 Occam 的 Razor 以审查项目计划是否可跟踪、是否合理、是否节省成本。从项目一开始到其整个生存期，全部的过程都是迭代的。

捕获总体情况

多范型设计中最困难的问题可能是：如何表示整个框架？首先必须有项目层次的架构，以定义子领域之间的接口和协议。如果系统部分在 C++中，部分在 C 中，部分在 Smalltalk 中，还有部分在 AOL 中，架构使用什么语言？Mary Shaw 和她的同事们已经对架构定义语言进行了研究，希望可以找到一种语言来满足这种需求[Shaw1994]。但这种语言在工业中尚未被广泛接受。仅仅依赖某一种设计记法是比较危险的，因为记法工件通常局限于它们的表达能力，对产品演进的跟踪并不很好，除非这些工件被放在代码生成过程中。我们可以在技术的最低层次上描述整个架构，接近于 C 函数绑定，但我们最为关注的系统结构就不太明显了。

最合理的选择是用项目的主要编程语言来描述系统的架构。如果项目基本上在 C++中，项目工具都支持 C++连接，我们就在 C++头文件中捕获架构。C++可以作为架构对话的混合语，覆盖多种范型——包括过程和模板。

当然，此文档只是结构文档的一方面。记法在系统开发中占有一席之地，此处不再详述。

另一种选择是从一个 CASE 工具（比如 ObjectTime[Selic+1994]）生成架构接口。CASE

工具易于受变化的微不足道的线性度的影响。也就是说，对架构的一点细微的改变可能会影响很多中间的设计工件，这样会导致返工（或者至少是重新编译）大量代码。一般情况下，CASE 工具不能很好地表达多种范型，还常常带来安全的错觉。很少有领域足够正规到可以用自动代码生成的程度。如果设计者确实发现了这样的领域，考虑为正式分析和效率建立一个 AOL 是比较明智的[Weiss1999]。

最后，C 语言或 IDL 这样的语言几乎总可用于定义系统各个部分之间的接口，因为大部分的技术和工具都可以与 C 进行连接。C 当然不是一种理想的模块定义语言，它缺乏可顺利捕捉架构抽象的特性。但因为它是可移植的、高效的语言，并且可以连接到很多工具和环境，所以它是形成系统各个部分之间接口的实用工具。有时，我们需要将特定数据类型转化成可移植的表示法（比如一个字符串），C 语言字符串可以直接被很多语言访问。

记法

设计者有时必须使用多种记法。数据库特别适合列表记法。数据库记法和面向对象的记法应当结合使用。如果可能，可以将不同的记法单独放在它们自身的子领域中。

还有一些其他的开发记法，比如用例[Jacobson+1992]、时序图、非功能性需求的自然语言描述，以及可以很好地服务于项目的形式规格说明。半符号工具（比如 CRC 卡[Beck1993]）建立在团队的领域专门知识基础之上，从而让面向对象的设计收敛于最佳的结构。当使用多种工具和技术时，减少工具、记法和语言的总量是很关键的。多种语言可以帮助开发团队的每一部分优化他们的表达力，但多种语言也会妨碍项目中团队之间的高效交流。培训和跨团队的成员关系提供了对这些问题的典型冲击，但最有效的解决方案是减少工具总量。

7.7.3 C++之外

尽管本书聚焦在 C++所支持的范型上，但设计者应当仔细考虑其他方案领域工具，每种工具都描述了一类重要的问题，它们与 C++的基本结构相差甚远：

- 数据库管理系统
- GUI-builders
- 分布式处理框架
- 常规商业语言（比如 Java Script）
- 为项目精心制作的的 AOLs 方法
- ……

设计者如果忽略了这些工具，将设计的所有方面都强制使用 C++实现，就会满足于减少工具的总量，结果可能错失了设计的可表达性、减少开发成本和可维护性的重要机

会。这些选择中有些本身就是编程语言；将多种编程语言混合起来是没有问题的。Vendor正逐渐认识到拥有更广泛支持能力的“多种语言”（multilanguage）的重要性。很多C++环境支持到Pascal和FORTRAN的连接，有些Eiffel环境支持到C++的连接，大部分的C、Pascal和FORTRAN程序都可以通过C语言绑定连接到C++（参见[Coplien1992]，附录A）。

在使用多种编程语言或工具集时，最好将这些语言紧密映射到子领域。每种语言或工具集都会逐渐形成自身的开发文化，保持这些开发文化的独立是很重要的。

7.7.4 领域专门知识

对于一无所知的家伙，即使你给了他工具，他仍然是一无所知。当代软件开发的一个典型缺陷就是期望用对象范型来弥补不成熟的领域知识。这个缺陷常常建立在这样一个幼稚的观点的基础之上：总有对象可供选择，问题词典和解决方案的类型之间存在强烈的“同构”（isomorphism）。通过对方案领域空间的探究，多范型设计摆脱了这种幼稚的观点。同时，多范型设计提供了共同性和差异性分析作为应用领域的工具，单独这些还不足以弥补不成熟的领域经验。系统架构工程师和设计者应当彻底地理解目标业务，并尽早对问题的作用域逐步形成统一的认识。如果整个团队缺乏关于应用领域的专门知识，那么面向对象的专门知识将会对你有所帮助。因为多范型设计显式聚焦在应用领域（和方案领域）上。它可以让设计者了解遗漏的信息，但却不能弥补设计者匮乏的开发专门知识。

方案领域专门知识也很重要。当使用多范型设计时（实际上任何设计都是如此），项目的成功依赖于所使用的各个范型的能力，以及设计者对这些范型的掌握。本书中所讲述的技术为理解用共同性和差异性就可以轻松描述的这些范型，以及理解某个实现中多种范型之间的相互作用打下了基础。但多范型设计并不像共同性和差异性表建议的那样呆板。好的设计总是要依靠尝试、洞悉力和经验的。

7.8 小结

本章我们详细解释了一个应用领域分析的示例，指明了如何从共同性分析得出C++设计和实现。分析的第一步是生成一个词典。下一步是将问题分成多个子领域，此过程非常依赖于设计者的经验和直觉。最后一步是具体情况的简单混合，其中单个子领域非常适合用某种特定的范型以及支持它的工具。

设计问题不会总是这么简单。本章中的示例是保证产生这样一种设计的准备，即每个领域可以清楚地用一种范型来实现。在这种情况下，子领域和范型的选择就不重要了。下一章我们会看到单个领域内紧密交织的范型的更一般情况。

第 8 章
将范型编织起来

本章介绍一些记法和技术，帮助编织组合领域内的范型，并将设计过渡到一个 C++ 实现。本章以一个运行文本编辑器示例为基础，详细说明了一个无限状态机抽象的分析和解决方案。

8.1 方法和设计

框架通常捕获了多个紧密耦合的子领域的设计（参见第 4.3.1 小节的论述）。大部分设计方法应用单个范型的原则，以减少软件模块之间耦合。当耦合无法避免或是有益处时，常规设计技术就无能为力了。有时可以通过对不同领域使用不同的范型来提高领域之间的耦合和内聚属性，第 7 章介绍了支持这种方法的技术。但有时可以通过组合领域内的多种范型来构建内聚最强和去耦最彻底的抽象。这种组合中有一些是简单而常用的，比如过程范型和模板的一般组合：

```
template <class T>
bool sort(T elements[], int nElements) {
    . . . .
}
```

或者，模板和基于对象的抽象的这种组合：

```
template <class T> class List {
public:
    void put_tail(const T&);
```

```
    T get_head();
    . . . .
};
```

我们想要知道如何通过共同性和差异性的提示来结合其他的 C++特性，以满足应用的需要。好的分析可以暗示除上述这些之外的常用范型组合。

有些设计甚至使用第 7 章介绍的多范型设计的划分技术也无法清晰地进行模块化。直觉业务领域可以展示相互的依赖性，因此也就带来了设计模块之间的耦合。本章探究了领域之间的循环依赖关系的多方面问题。

8.2 共同性分析：共同性维度是什么？

我们已经分析了应用领域和方案领域中的共同性和差异性。我们用领域分析，并从手边的应用抽象扩展到通用的业务抽象，以找到应用子领域中的共同性。在方案领域中，我们寻求可用技术和范型中的共同性。我们已经将 C++及其范型作为解决方案技术。并且可以对分析进行扩展，以容纳其他方案领域（如果需要的话）。

设计是将应用领域结构同适当的方案领域结构结合起来的过程。换句话说，我们必须将问题的共同性和差异性与约束此解决方案的内容结合起来。第 7 章介绍了一对一简单映射关系的示例——设计者所选择的某单个范型操纵着各个子领域。

单一的划分很难捕获一个复杂系统的所有重要结构。复杂性与一个系统的不同的且有意义的划分（或者视图）的数量成正比。在有些系统中，不仅对象范型不太适用，任何单一的范型都难以很好地适用。要说明这一点并不容易，因为必须证明特定类型的架构不存在。然而，暂时假设对象范型非常适合我们所熟知的某些系统，并且有一些在对象范型出现以前就曾经构建过类似系统的程序员（可能读者本身就属于此列）。在那个时候，程序员本可以组合这些范型（可能是数据抽象、数据函数模块性和函数指针）来解决设计问题。现在当然也还有这样的系统，并且有些系统难以用单个范型充分地表达系统结构的共同性和差异性。

有时我们比较幸运，范型划分与直觉的子领域划分是一致的。在这种情况下，第 7 章所讲述的技术就非常适用——我们在各个子领域或子系统层次上应用范型，而不是在系统层次上应用范型。但有时我们发现：每个子领域仅仅使用一种范型是无法进一步细分直觉的子领域划分的，必须同时使用多种范型才能捕获和管理这种复杂性。

我们来考虑通用文件抽象的设计。系统设计可能包括索引顺序文件、分块文件和其他文件类型。这些文件具有共同的行为，各自的实现不同，这就暗示我们将继承作为实现技

术。但单独的文件抽象可以因其所支持的字符集（char、unsigned char、wchar_t或用户自定义字符集类型）不同而不同。这些差异性表现在文件族成员的接口中。这是一种不同类型的差异性，一种相对于共同实现结构背景的接口中的差异性，这就暗示我们应当使用模块作为实现技术。如果两个应用差异性可以很好地相互结合，则我们可以使用第7章所介绍的技术来处理这个问题。例如，如果所有的索引顺序文件使用wchar_t，而其他所有的文件使用 char，则实现就很直接。但字符集划分的边界与文件类型划分的边界就不一致了。当然，字符集和文件类型可以在任何一种组合中进行混合。

我们必须将这两种范型以某种形式编织在一起。单独一种范型无法捕获生成所有族成员所必需的差异性。设计者必须根据差异参数的原始组合来生成族成员。我们将在第8.3节中用文本编辑器的 **Text Buffer** 子领域来探究这个问题。**Text Buffers** 形成了一个族，但族成员各不相同（在复杂的形式上不同，复杂独立性反映在差异参数上）。我们不能使用第 7 章所介绍的方法来寻找捕获其结构的单个范型。将子领域划分成更小的部分也无济于事（因为缺少反例，我们很难证明这一点，所以暂时将此缘由作为练习留给读者）。我们可以识别合乎需要的范型，并逐步展开应用它们的线索。

完成这些工作以后，我们将进入到第 8.4 节来解决更高级的问题：对多个相互依赖的子领域的多范型设计。在这一节中，我们将回过头来研究 **Text Buffers** 和 **Output Medium** 这两个领域之间的关系（这两个领域的设计是相互交叉的）。我们知道领域是永远不可能完全独立于其他领域的——毕竟它们的代码是作为一个整体为系统的工作服务的。当我们需要充分利用划分技术时，任意两个领域之间的耦合都可以相差很大。有时耦合比较强，进一步划分为子领域难免有些武断。当直觉提示我们这些子领域应当予以保留，以便服务于长期的维护时，甚至当这些子领域紧密耦合到其他子领域时，就需要一种方法来管理紧密耦合的各个子领域之间关系。

8.3 一组共同性中的差异性的多个维度

我们可以从文本编辑器的设计中找出一个例子，来说明超出第 7 章所介绍的简单问题或解决方案的多范型设计的复杂性。在本节中，我们将独立地分析 **Text Buffer** 子领域，以此说明如何在单个领域中使用多种范型。

8.3.1 差异性分析

我们是根据领域的共同性来形成和描述抽象的。这里，我们来考察 **Text Buffer**，它是文本编辑器中编辑文件期间用于保留文件内容的逻辑副本的一个抽象族。（确切来讲，

Text Buffers 是 **Text Editing Buffers** 的泛化(第 3 章),适合于文本编辑以外的很多应用。)文本文件可以驻留在磁盘上。文本缓冲区表示文本文件到编辑程序的状态为止。它的高速缓冲变化,一直持续到用户命令编辑器将文本缓冲区中的内容转储到磁盘文件中。一个简单的 **Text Buffer** 将保留整个文件的一个主存储器副本。更高级的 **Text Buffer** 可以实现分页调度或交换方案,从而在保持磁盘文件内容的完整驻留副本的同时有效利用主存储器。再高级一些的 **Text Buffers** 可以支持回滚、简单的版本更新方案或并行多用户编辑。

这些抽象形成了一个族,使之成为一个族或子领域的是所有的变量共享相同的行为。也就是说,他们可以从一个正在编辑的图像中获得数据,或根据需要替换一个正在编辑的图像中的数据。所有的文本缓冲区还共享着某些结构:所编辑图像中的行数和字符数的记录、当前行号等。

族成员通过它们的差异参数各自的值相互区别。我们用表 8.1 所示的差异性表捕获族差异性。表中的第一列列出了差异性的参数或值域。第二列是说明。第三列列出了属于此值域内的值。第四列列出了绑定时间。最后一列指定了默认值。

表 8.1 文本编辑器的共同性领域(Text Buffer)的差异性分析(共同性:行为和结构)

差异参数	意 义	领 域	绑 定	默认值
Output Medium(输出介质)	文本行的格式对输出介质敏感	数据库、RCS 文件、TTY、UNIX 文件	运行时间	UNIX 文件
Character Set(字符集)	不同缓冲区类型应支持不同的字符集	ASCII、EBCDIC、UNICODE、FIELDATA	源码时间	ASCII
Working Set Management(工作区管理)	不同应用需要在内存中高速缓冲不同数量的文件	整个文件、整个页面、LRU 确定	编译时间	整个文件
Debugging Code(调试代码)	内部开发中应当有调试中断,并应当永久保留于源代码中	调试、产品	编译时间	产品

随着过渡到多范型设计,领域和差异参数的值域之间的交叠逐渐变得重要。第 7.3.2 小节中首先引进的差异性依赖关系图帮助我们直观显示了这种依赖关系。我们可以直接

根据差异性分析表中的信息来构建这样一个图，如图 8.1 所示。箭头从相关的领域抽象起始，至表示差异参数的椭圆框为止。该图看起来可能像一个摇动木马，但它却能很好地帮助我们。

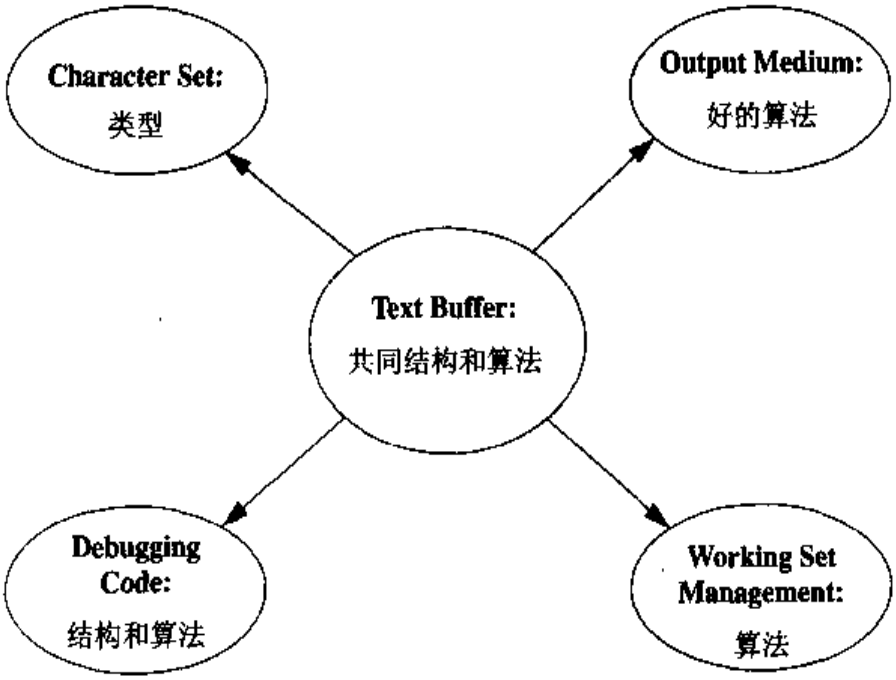


图 8.1 **Text Buffer** 共同性分析的差异性依赖关系图

8.3.2 在 C++中表达共同性和差异性

我们在第 3 章中细致讨论了差异性，现在要了解如何表达相对于共同性背景的差异性，我们可以考察如何使用 C++结构来表达设计共同性和差异性。将 **Text Buffer** 差异性分析的结果与设计者可以在 C++中表达的抽象结合起来，就可以实现这一目标。

共同性描述了一个族的特性，差异性区分出族成员。C++可以表达几种形式的共同性和差异性。通过考虑表 8.1 所示的设计共同性和差异性，并将它们与表 6.2 中总结的 C++的共同性和差异性相匹配，就可以选择出适当的 C++语言结构。该表将共同性维度、差异性维度和绑定时间映像到 C++语言特性。绑定时间表达了编码周期的最初阶段，差异性参数可以在这一阶段进行绑定。例化列说明是否允许同一个抽象存在多个示例。如果允许，就假定状态中的差异性。最后一列为给定的共同性、差异性和例化指明了最佳的 C++结构。

我们可以查看表 8.1 中所捕获的结构需求，并将它们同表 6.2 中所捕获的结构表达结合起来，从而了解什么样的 C++语言特性最适合设计的不同部分。我们用一个注释版本的差异性分析表来捕获新得到的这些情况，结果如表 8.2 所示。表中添加了三种类型的注

释（用斜体区分）。第一，我们记住共同性领域是 **Text Buffers**。认识到文本编辑缓冲区的特征是通过操作（行为）和结构的共同性来描述的。第二，在“差异参数”列中，我们通过 C++共同性表中的差异性类别来描述值域的特性。通过查看值域变量的差异性，并从表中直接看出绑定时间，我们就可以从 C++共同性表中选择一种适当的技术。这种技术作为注释写在此表的“默认值”列中（并不是因其属于这一列，而是为了节省表格中的列）。

表 8.2 文本编辑器的共同性领域（Text Editing Buffers）的转变分析

差异参数	意 义	领 域	绑 定	默认值
Output medium（输出介质） *结构* *算法*	文本行的格式对输出介质敏感	数据库、RCS 文件、TTY、UNIX 文件	运行时间	UNIX 文件 *虚函数*
Character set（字符集） *非结构的*	不同缓冲区类型应支持不同的字符集	ASCII 、EBCDIC 、UNICODE、FIELDATA	源码时间	ASCII *模板*
Working set management（工作区管理） *算法*	不同的应用需要在内存中高速缓冲不同数量的文件	整个文件、整个页面、LRU 确定	编译时间	整个文件 *继承*
Debugging code（调试代码） *代码段*	内部开发中应当有调试中断，并应当永久保留于源代码中	调试、产品	编译时间	产品 `#ifdef`

共同性分析表明我们应当对输出使用虚函数（因为差异性参数 **Output Medium** 的不同值需要不同函数）。结构和总算法中的差异性必须用继承，编译时间对其他字符集的依赖性适合用模板。

通常，我们将继承当作是“一种类型”的关系。设计中的各种文本缓冲区的继承是不同的。我们可以非正式地讨论“分页文本缓冲区”和“LRU 缓冲区”，因此建议将内存

管理算法当作族成员的主要的区分。表 8.2 中“工作区管理”这一行也可以改写成“文本缓冲区类型”。如果使用了继承就会出现这种情况。实际上，我们可能会在错误的假定（存在主要的区别）上过于频繁地使用继承。

结果代码可能为：

```
template <class CharSet>
class OutputFile {
public:
    virtual void write(
        const class TextBuffer<CharSet> &);
    . . . .
};

template <class CharSet>
class TextBuffer {
public:
    TextBuffer(const OutputFile<CharSet> &);
    basic_string<CharSet> getLine(LineNumber) const { }
    void insertLine(LineNumber, const string&) { }
    void deleteLine(LineNumber) { }
    . . . .
};

template <class CharSet>
class WholeFileTextBuffer: public TextBuffer<CharSet> {
public:
    . . . .
    WholeFileTextBuffer(const OutputFile<CharSet> &);
    basic_string<CharSet> getLine(LineNumber l) const
        { . . . . }
    void insertLine(LineNumber l,
        const basic_string<CharSet>&s){
        . . . .
    }
    void deleteLine(LineNumber l) { . . . . }
    . . . .
};

template <class CharSet>
class LRUTextBuffer: public TextBuffer<CharSet> {
public:
    . . . .
```

```
    LRUTextBuffer(const OutputFile<CharSet> &);
    basic_string<CharSet> getLine(LineNumber l) const
        { . . . . }
    void insertLine(LineNumber l,
        const basic_string<CharSet>&s){
    #ifndef NDEBUG
        . . . .
    #endif
        . . . .
    }
    void deleteLine(LineNumber l) { . . . . }
    . . . .
};
```

用转变分析表对代码进行的粗略分析足以让有经验的 C++设计者相信：二者都表示相同的设计。但是，我们没有提供从该表到相应代码的自动转换。编码通常是直接进行的，同时，好的设计通常知道何时通过难以编码的模式或某种语言特性来表达这种转换。多范型设计没有提供这种自动转换，这正好为创造性的理解留出了余地。

8.4 相互依赖的领域

第 7.1.2 小节讨论了 MVC 的例子，聚焦在两个典型的用户接口设计之间的依赖关系上。我们找到了一些循环依赖关系，**Controller** 依赖于 **View** 的结构，以了解屏幕的布局（光标是否在某个按钮上？）**View** 依赖于 **Controller**，以支持类似高亮这样的功能性。我们称这些领域是相互依赖的。第 7 章所介绍的方法并不足够强大，它无法处理这种循环依赖关系。实际上，要规范化循环依赖关系的解决方案是比较困难的。但大部分解决方案在某个依赖性方向上依赖于我们正在使用的某个范型（比如继承），而在相反的依赖性方向上又依赖于另一种范型（比如模板或通过指针实现的运行时选择）。本章后面的内容将对这个问题进行一些分析，并进一步展开文本编辑的示例。

第 8.3 节的分析适用于孤立的 **Text Buffers**。要获得更广泛的系统分析，我们必须在与之相互作用的其他抽象的语境中研究 **Text Buffers**。在对文本编辑器的分析中，我们发现 **Output Medium** 是文本编辑的又一个领域。我们对其进行单独的差异性分析，并得到图 8.2 所示的差异性图。然后我们用 C++差异性表来进行转变分析，得到表 8.3。

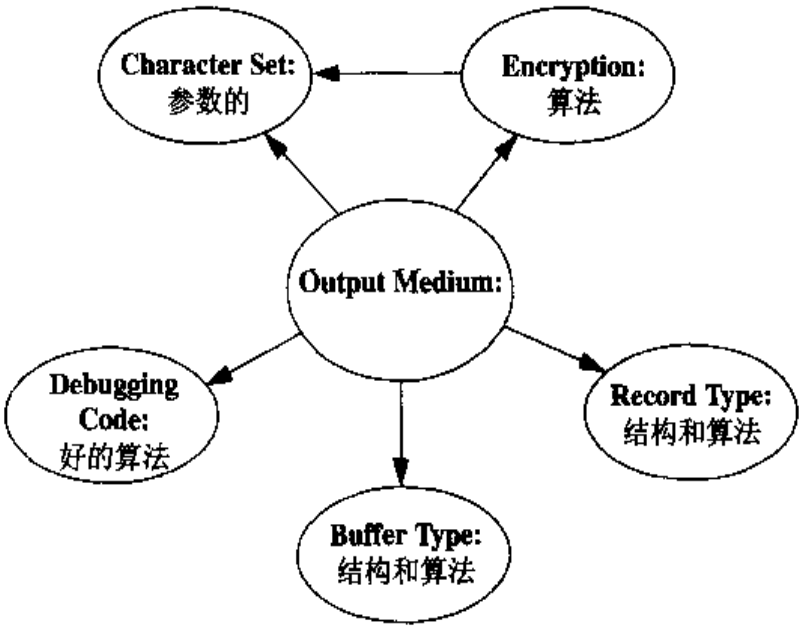


图 8.2 **Output Medium** 共同性分析的依赖关系图

表 8.3 文本编辑器共同性领域（Output Medium）的差异性分析

（共同性：多种行为，建议使用继承）

差异参数	意　义	领　域	绑　定	默认值
Buffer type （缓冲区类型） *结构* *算法*	文本行的格式和处理对 Text Buffer 类型敏感	检查点，分页的、有版本的、完整的文件……	编译时间	完整文件 *继承*
Record type （记录类型） *结构* *算法*	针对不同记录类型的不同格式化输出	数据库、RCS 文件、TTY、UNIX 文件	运行时间	字符流 *虚函数*
Character set （字符集） *非结构的*	不同文件类型应支持不同的字符集	ASCII、EBCDIC、UNICODE、FIELDATA	源码时间	ASCII *模板*
Encryption （加密） *算法*	不同的市场需要不同的加密技术	无，PGP（非常好的隐私保护）， NBS（美国国家标准局）	编译时间	无 *继承*
Debugging code （调试代码） *代码* *代码段*	内部开发中应当有调试中断	调试、产品	编译时间	产品 `#ifdef`

图 8.2 反映了设计过程中的简化操作。我们用此图来捕获作为差异参数的 **Character Set** 的依赖性，而不是单独画出 **Encryption** 的差异性图。要实现这个目标，首先需要认识到 **Encryption** 不是一个“初始”领域（我们将 **Character Set** 当作初始领域），其次要认识到它依赖于同一个差异性图中的其他领域。只有至少用一个差异性图捕获了所有这种依赖性以后才允许使用这种简化操作。所有图表的最后组合将捕获和优化这种依赖关系。

Output Medium 表为 **Text Buffers** 承担了到差异性分析表的重要关系。二者在运行时都依赖于同一个领域，这个领域属于文件或记录类型（这两种类型紧密关联）。这种依赖关系是可逆的。由于 **Text Buffer** 差异参数领域对 **Output Medium** 的值域存在依赖性，以及 **Output Medium** 差异参数领域对 **Text Buffers** 的存在依赖性，所以这个问题比较复杂。这就意味着在多个维度中，结构和算法的选择依赖于运行时出现的文件/记录类型和缓冲区类型。这样的复杂设计中一般存在着任意多种依赖性（或者说依赖关系）。我们无法在单独子领域的基础上选择范型。相反，我们还必须解决子领域之间接口的问题。

使用多重调度（[Coplien1992]中的“multimethod”）可以部分解决这个设计问题。多重调度是某些面向对象的编程语言（比如 CLOS）的一种特性，它让运行时的方法寻求使用多个方法参数。C++成员函数只使用了一个参数进行方法选择：地址被作为 `this` 隐式地传递给成员函数的参数。尽管 C++不直接支持多重调度，但设计者可以用适当的表达方式来模仿“多方法”（multimethod）[Coplien1992]。但多重调度只表达出算法中的差异性，没有表达出结构中的差异性。对于缓冲区设计，必须探究其他的解决方案。

图 8.2 **Output Medium** 的差异性依赖关系图仅仅捕获了此子领域中的差异参数（但并没有捕获 **Encryption** 和 **Character Set** 之间的明显依赖性）。我们已经在图 8.1 中为 **Text Buffers** 建立了一个简单的关系图。

第 7.3.2 小节中在介绍领域依赖关系图时，我们预先考虑了这里正在讨论的用法。我们谈到图中的差异参数本身可作为领域来解释，就像中间的子领域椭圆框那样。如果我们相信某种设置值域可以作为另一种设计的领域，那么这就是有直觉意义的。实际上，我们将差异性表（如表 8.3 所示）中的每个差异参数都当作“领域”来讨论。但差异参数不需要是 `char` 和 `int` 这样的初始领域，它们是我们已经进行过应用领域分析的领域。一般而言，认为差异参数是与它们自身的领域紧密联系总是有益的。

注意图 8.1 和图 8.2 共享了一些节点。每个图在各自表示其子领域的结构的同时，两个图的“联合”也表示两个子领域的结合。每个图中间的领域椭圆框作为差异参数出现在另一个图中。我们将这两个子领域差异性依赖图合并成单个的图（如图 8.3 所示）。

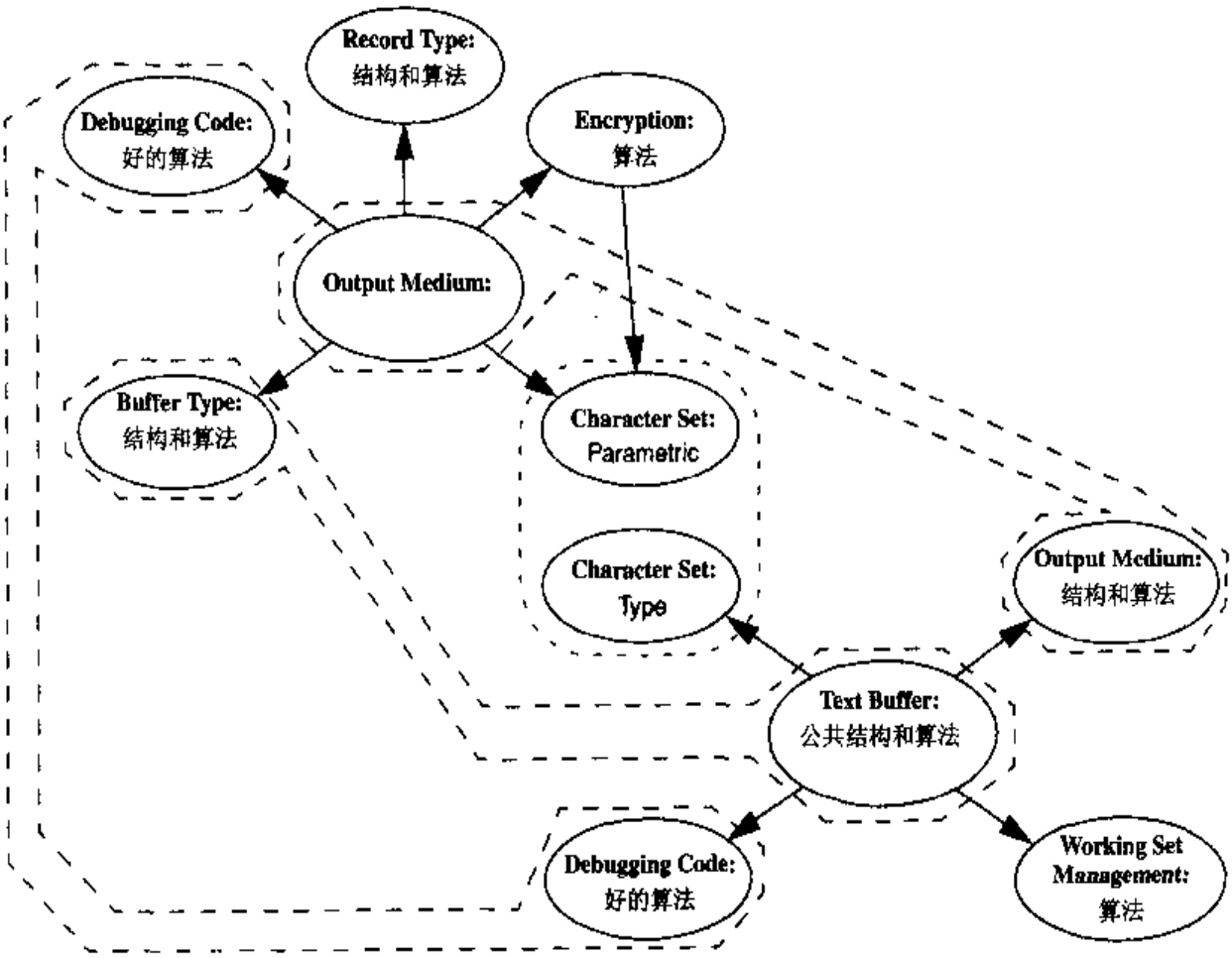


图 8.3 合并两个应用领域

图 8.3 明确显示了来自两个子领域图中的复制节点。必须注意只合并图中的兼容节点。例如，我们知道可以合并两个 **Character Set** 类型的节点，因为 **Text Buffer** 和 **Output Medium** 领域在 **Character Set** 类型中是协变的。也就是说，两个较大的领域中此差异参数的类型相同。它们都相对于此差异参数而变化，并且两个领域中此差异参数的绑定时间是相同的。如果应用需要在运行时间将 **Character Set Type** 绑定到 **Output Medium**，并在运行时间绑定到 **Text Buffer**，这两个节点就不能合并。这种记法无法捕获这个细节层次，这要留给设计者处理。如果项目要为架构文档使用差异性依赖关系图，就需要增加记法来捕获绑定时间和其他相关概念。

我们将超节点（虚线表示的）中的重复节点圈起来，然后在图 8.4 中简化或重新定制此图。该图描述了完整的设计（在这一点上），包含了 **Output Medium** 和 **Text Buffer**。当 **Debugging Code** 同时作为输出介质和文本缓冲区的属性出现时，两个代码集可能是无关的（它们是不同的领域），所以此图中不能将它们合并起来。

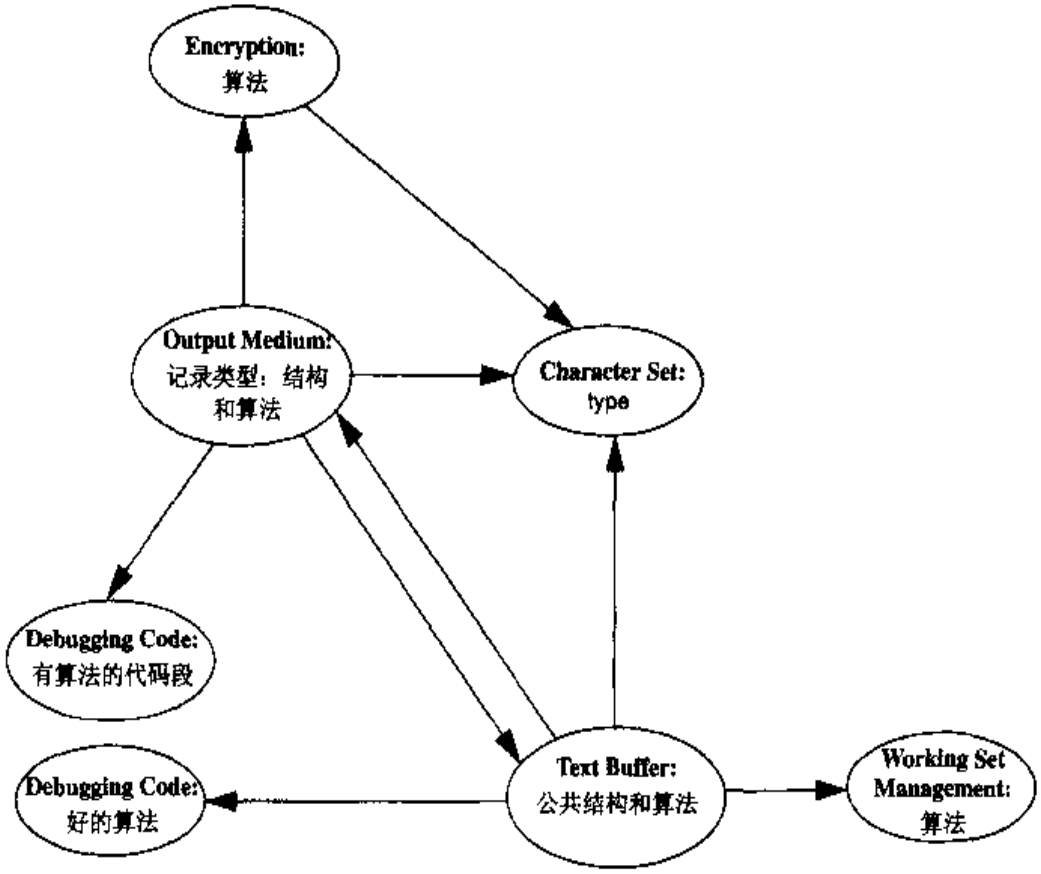


图 8.4 简化后的差异性依赖关系图

这个新的架构描述了一个新的设计空间，它拥有自己的共同性和差异参数。我们可以将其当作一个从其父空间那里继承了复杂混合染色体的新事物。这个新的设计空间明显不同于其父空间，但它展示了两个父空间的特性。我们可以将此图作为这两个子领域合并后的架构。注意，合并后的图中各个子领域架构都不明显。复杂的依赖性迫使我们从更宽的角度得出一个新的结构，此结构有自己的“身份”和“个性”。

有些抽象（比如字符集）可以令人满意地解析出来。有些参数对它们的原始子领域来说是惟一的，比如工作区管理和加密算法。但我们发现每个领域的核心抽象都相互将对方作为差异参数。文件类型导致 **Text Buffer** 中相对于行为和结构中的共同性背景的类型差异，还导致了 **Output Medium**（此领域的读、写等行为共同的）中结构和算法的差异性。我们可以对回路中的每个箭头使用继承，当然，相互派生是不可能的。

先假设需要将所有的绑定决定推迟到运行时间进行。如果每个抽象都有一个类型字段，并且每个抽象都使用对其他抽象的用例分析调整了自身的行为，那么我们可以在 C++ 中实现这个设计。即使 C++确实有多重调度，我们也无需承受成熟的运行时的灵活性的复杂和低效，除非应用需要。如果两个领域的主要抽象是相互依赖的，就可以使用绑定

时间来选择一种适当的技术。

在后面几节中，我们将审视此架构的三种实现以及每种实现中的一些细微差异。每种实现都略微不同地满足了对绑定时间的需要。

8.4.1 第一种情况：运行时绑定

我们先假设所有的设计决定都可以在运行时间绑定，并且我们可以使用支持运行时绑定的语言特性。这些语言特性包括模板、继承（不带虚函数）和重载。Barton 和 Nackman[Barton+1994]常常将前两个特性混合起来，以打破这里发现的循环依赖回路。这是解决相互依赖的领域问题的重要代码模式。此技巧的实质是使用下列形式的代码：

```
template <class D> class Base {
public:
    . . . . D* . . . .  // a reference to D through
                        //              a pointer or
                        //              reference
    virtual D *base1() { . . . . }
    void base2() { . . . . }
};

class Derived: public Base<Derived> {
    . . . .
    Derived *base1() { . . . . base2() . . . . }
};
```

也就是说，Base（基类）将其自身的 Derived（派生）类当作模板参数。这样，Base 对 Derived 的了解就有可能比它仅从自身作为 Derived 的基类的角色所推断出来的内容要多。（实际上，基类通常不会注意它们的派生类的存在，这是 C++语言的重要设计特性。）特殊情况下，Base 模板的签名预见了 Derived 类的类型（即在成员函数 base1 的返回类型中）。

这种形式值得仔细研究。建议读者试着使用一种一般的记法来图解这种形式的代码，或者试着使用这种方法来编写一个示例，从而帮助自己逐步形成对此种方法的衍生方法的直觉。

所编写的代码的形式与此非常相像。下面是一个普通的 OutputMedium 模板。我们捕获对 TextBuffer 的依赖性，将其作为一个模板参数：

```
template <class TextBuffer, class CharSet>
class OutputMedium {
public:
```

```
    void write() {
        . . . .
        subClass->getBuffer(writeBuf);
    }
    OutputMedium(TextBuffer *sc): subClass(sc) { }
protected:
    TextBuffer *subClass;
    CharSet writeBuf[128];
};
```

从这个模板派生出来几个类，其中一个为：

```
template <class TextBuffer, class Crypt, class CharSet>
class UnixFile: public OutputMedium<TextBuffer, CharSet>,
    protected Crypt {
    // inherited as a mix-in
public:
    UnixFile(TextBuffer *sc, basic_string<CharSet>
        key = ""):
        OutputMedium<TextBuffer, CharSet>(sc),
        Crypt(key) { }
    void read() {
        CharSet *buffer;
        . . . .
        Crypt::decrypt(buffer);
    }
};
```

加密是一个混合类，它与保护继承结合在一起。我们不想让 UnixFile 的客户直接访问 Crypt 成员函数，但又想让他们可以访问 UnixFile 的派生类（如果存在）。在简单混合类中，可以这样进行加密：

```
template<class CharSet>
class DES {
protected:
    void encrypt(basic_string<CharSet> &);
    void decrypt(basic_string<CharSet> &);
    DES(basic_string<CharSet> key);
};
```

我们也可以从下面这个模板扩展出多个 TextBuffer 基类，每个基类对应于一个字符集类型：

```
template <class CharSet>
class TextBuffer {
```

```
public:
    basic_string<CharSet> getLine() {
        basic_string<CharSet> retval;
        . . . .
        return retval;
    }
    void getBuffer(CharSet *) { . . . . }
    TextBuffer() { . . . . }
};
```

这里，我们将这两种特性的结合放在 TextBuffer 派生类中，并使用派生捕获对 UnixFile 的依赖性：

```
template <class Crypt, class CharSet>
class UnixFilePagedTextBuffer: public
    TextBuffer<CharSet>,
    protected UnixFile<UnixFilePagedTextBuffer<Crypt,
        CharSet>, Crypt, CharSet> {
public:
    UnixFilePagedTextBuffer():
        TextBuffer<CharSet>(),
        UnixFile<UnixFilePagedTextBuffer<Crypt,CharSet>,
            Crypt, CharSet>(this) {
        . . . .
    }
    basic_string<CharSet> getLine() {
                . . . .
        read();    // in UnixFile
        . . . .
    }
};

int main() {
    UnixFilePagedTextBuffer<DES<wchar_t>, wchar_t>
        buffer;
    basic_string<wchar_t> buf = buffer.getLine();
    . . . .
}
```

此解决方案展示了具体领域的代码之间的强烈耦合，尤其是 UnixFilePagedTextBuffer 模板与其参数之间的耦合。使用运行时绑定可以更简单地为具体领域的代码去耦。

我们还可以将 DES 加密类作为参数，从而把 CharSet 作为参数（由于比较简单，这里不再赘述）。

8.4.2 第二种情况：在运行时间 Buffer Type 依赖于 Output Medium 类型；在编译时间 Output Medium 类型依赖于 Buffer Type

在我们的示例中，文本缓冲区的行为依据输出介质类型的不同而不同，并且使用哪种行为的决定必须一直推迟到运行时间。输出介质类型依赖于缓冲区的类型，但可以在编译时间绑定此类型的决定。我们使用多重调度代码模式来解释这个问题。文本缓冲区告知其相关输出介质它要执行某项操作，然后输出介质给予应答。另外，使用多重调度代码模式来管理行为中的差异，我们就可以使用继承来管理结构中的差异性。结构的选择要遵循算法的选择，所以我们获得了与对算法进行多重调度相同的结构灵活性。注意在这种情况中，有些结构（输出介质类型的结构）是运行时绑定的，尽管算法选择结构与多重调度代码模式的选择结构相同。

我们注意到：在运行时间，输出介质类型依赖于缓冲区类型。领域是 **Output Medium**，差异参数是缓冲区类型（类型中的差异），**Output Medium** 的基本共同性是结构和算法。表 6.2 表明：应当使用模板。我们可以用一种比较高效的方式使用模板和继承来打破依赖性循环。Barton 和 Nackman[Barton+1994]先后对这两种语言特性的很多应用进行了改进和分类。我们可以对图标循环中的某个方向的箭头使用模板，对相反方向的箭头使用继承。在这个示例中，我们还是用继承来处理加密问题（后面会有更多的介绍）：

```
template <class CharSet, class ATextBuffer>
class OutputMedium {
public:
    virtual void write(const ATextBuffer *) = 0;
private:
    virtual void encrypt(basic_string<CharSet> &s) {
    // default encryption:  none
    . . . .
};
```

输出介质（**Output Medium** 领域）也依据记录类型的不同（是否为 Unix 文件、数据库、RCS 文件或其他类型）而不同。这种差异表现在结构和算法中。需求要求在运行时间跟踪差异性，所以我们使用继承和虚函数。虚声明出现在上面的基类中。成员函数 `write` 将在本章后面讨论。

```
template <class CharSet, class ATextBuffer>
class Unixfile: public OutputMedium<CharSet, ATextBuffer> {
public:
    // compile-time bound:
    void write(const ATextBuffer *buf)
```

```
        { buf->unixWrite(this); }
    UnixFile(string file): fileName(file) { /* . . . . */ }
    . . . .
private:
    string fileName;
    . . . .
};
```

文本缓冲区在编译时间依赖于字符集，在运行时间依赖于输出介质。例如，文本缓冲区可以利用RCS文件中的版本信息，或使用数据库介质中的永久标记或行标识符。在运行程序时，我们可能想要将缓冲区中的内容写入到不同的输出介质中。输出介质同时导致了文本缓冲区的结构和算法中的差异性。我们使用多重调度代码模式的一个差异来处理这个问题[Coplien1992]。当调用TextBuffer<CharSet>的成员函数write时，它会将其相关的输出介质（未知类型）发送到适当的 write 函数。OutputMedium 通过调用 TextBuffer <CharSet>适当的成员函数强制进行接收（正如上面在UnixFile::Write中一样）。

```
template <class CharSet, class DerivedClass>
class TextBuffer {
    // DerivedClass is passed in just to support
    // casting, e.g., from
    // TextBuffer<char,PagedTextBuffer<char> >*
    // to PagedTextBuffer<char>*. Because the binding is
    // static, we know the downcast is safe
public:
    void write(OutputMedium<CharSet, DerivedClass> *op)
        const {
        op->write(static_cast<DerivedClass*>(this));
    }
    void unixWrite(OutputMedium<CharSet, DerivedClass> *)
        const;
    void databaseWrite(OutputMedium<CharSet,
        DerivedClass> *)
        const;
    . . . .
};
```

输出介质类型中的差异性也派生出结构中的差异性。这捕获在输出介质类中，而不是在它自己的文本缓冲区类中。每对缓冲区类型和输出介质专用的算法出现在对应的类（针对缓冲区类型而命名）的成员函数（针对输出介质而命名）中。

```
template <class CharSet>
class PagedTextBuffer: public TextBuffer<CharSet,
        PagedTextBuffer<CharSet> > {
```

```
public:
    void unixWrite(OutputMedium<CharSet,
        PagedTextBuffer<CharSet> > *theMedium) const {
        . . . .
    }
    void databaseWrite(OutputMedium<CharSet,
        PagedTextBuffer<CharSet> > *theMedium) const {
        . . . .
    }
    . . . .
};
```

TextBuffer<CharSet>的每个派生类都有各自的实现（unixWrite、databaseWrite和针对具体输出介质类型的其他函数）：

```
template <class CharSet>
class FullFileTextBuffer: public TextBuffer<CharSet,
        FullFileTextBuffer<CharSet> > {
public:
    void unixWrite(OutputMedium<CharSet,
        FullFileTextBuffer<CharSet> > *theMedium) const {
        . . . .
    }
    void databaseWrite(OutputMedium<CharSet,
        FullFileTextBuffer<CharSet> > *theMedium) const {
        . . . .
    }
    . . . .
};
```

正如领域分析所规定的那样，在编译时间，一个输出介质与一个文本缓冲区是相互关联的（在运行时间带有一个对应的文本缓冲区实例）。注意，对于这个main程序，尽管所有输出介质的目标代码都有了，但FullFileTextBuffer的目标代码却没有结合成可执行程序：

```
int main() {
    PagedTextBuffer<char> textBuffer;
    UnixFile<char, PagedTextBuffer<char> > *file =
        new UnixFile<char,
            PagedTextBuffer<char> >("file");
    textBuffer.write(file);
    . . . .
}
```

我们可以用继承来处理加密的问题。这将直接从 UnixFile<char,PagedText

Buffer<char>>派生出一个新的类，覆盖成员函数 encrypt，从而创建一个支持我们所选择的加密算法的新的族成员。如果加密算法的数量有限，它们可以存储在一个过程库中，从派生类中被覆盖的 encrypt 函数来进行适当的调用。当然作为另外一种选择，我们也可以使用“混合”（mix-in），以一种更加“面向对象的方式”来处理加密的问题。混合是一种继承用来为一个派生类提供服务的“轻便”类。混合通常是私有或保护的，并且常常作为几个基类中的一个出现：

```
template<class CharSet>
class DES {
protected:
    void encrypt(basic_string<CharSet> &cp) { . . . . }
    void decrypt(basic_string<CharSet> &cp) { . . . . }
    DES(basic_string<CharSet> key) { . . . . }
};
class DESUnixFile:
    public UnixFile<char, PagedTextBuffer<char> >,
    protected DES<char> {
public:
    void encrypt(string &s) { DES<char>::encrypt(s); }
    . . . .
};

int main() {
    PagedTextBuffer<char> textBuffer;
    DESUnixFile file("afile");
    . . . .
    textBuffer.write(file);
    . . . .
}
```

当然，只有应用允许加密在编译时和源码时绑定，在 C++中这种方法才可用。在带有动态多重继承的语言（比如 CLOS）中，它可以在运行时绑定。

可替代的解决方案

C++提供了多种语言特性来表达类似的绑定时间和差异性类别。这里，我们要研究前面介绍过的解决方案的差异。

领域分析规定加密应当在编译时绑定。如前所述，混合是实现这个目标的一种方法。继承层次结构是运行时绑定的，并且它可以捕获行为中的差异性（也可以捕获结构中的差异性，但这对差异参数 **Encryption** 来说并不重要）。

其他的 C++结构也可以很好地表达同种差异性，模板就是这样的一个例子。我们可

以将加密算法作为模板参数，它可以被相应的类访问（这里指 OutputMedium），最终获得与前面我们使用多重继承时相同的结果。可以这样声明 OutputMedium：

```
template <class CharSet, class TextBuffer, class Crypt>
class OutputMedium {
public:
    virtual void write(const TextBuffer &) = 0;
    . . . .
};
```

从 OutputMedium 的一个实例派生而来的 UnixFile 也带有加密参数，并将其传递给它的基类。（其他选择是将所有的加密工作局限在一个 UnixFile 派生类中完成，在这种情况下，Crypt 参数将从 OutputMedium 模板中删除。）

```
template <class CharSet, class TextBuffer, class Crypt>
class UnixFile: public OutputMedium<CharSet, TextBuffer,
    Crypt> {
public:
    UnixFile(const string &filename);
    ~UnixFile();
    void write(const TextBuffer &buffer) {
        Crypt c = ::getKey();
        basic_string<CharSet> s;
        . . . .
        c.encrypt(s)
        . . . .
    }
    . . . .
};
```

为方便起见，加密算法本身被打包为某个类中的一个静态函数，或者是更加通用的函数符（[Gamma1995]的 **Strategy** 模式）：

```
class DES_char_Encryption {      // functor
public:
    void encrypt(string &);
    void decrypt(string &);
    DES_char_Encryption(const string &);
};
```

有些加密算法（比如“流码”，stream cipher）拥有局部状态。这就暗示函数符需要以静态函数来实现。

与前面一样，PagedTextBuffer 可以从一个 TextBuffer 实例派生而来，主要的代码与前一个例子中的类似。有一点例外，这里的加密差异性作为一个捕获了差异参数

的显式模板参数出现，这比以模板自身的名字来编码加密算法更具审美愉悦性，但我们依然保留了编译时绑定的相同效率：

```
int main() {
    UnixFile<char, PagedTextBuffer<char>,
        DES_char_Encryption>
        file("Christa");
    PagedTextBuffer<char> textBuffer;
    . . . .
    textBuffer.write(file);
    . . . .
    return 0;
}
```

对于绑定时的细微差异，设计的选择有很多。例如，文件结构可以将一个 TextBuffer 引用作为一个参数，从而在创建文件时将文本缓冲区的类型（和实例）绑定到文件，而不是在每次调用 write 时重新进行绑定。这种形式和前面所讲的形式都使用了运行时绑定。如果我们将缓冲区当作一个参数传递给 write 函数，基于构造函数的绑定就会早一些。我们可能只对每个文件实例进行一次基于构造函数的绑定。这是多范型记法所支持的更加精细的绑定粒度，但 C++对此支持得也很好，因此，它是一种有效而重要的设计考虑因素。

8.4.3 第三种情况：在运行时间 Buffer Type 依赖于 Output Medium Type，在运行时间 Output Medium Type 依赖于 Buffer Type

我们可能想让文本编译器能用任意文件动态地配置任何的缓冲区类型，并在最后时刻根据语境（可用内存、用户偏好等）在抽象中进行选择。这种情况需要充分的运行时的灵活性，两个核心的抽象在运行时间相互依赖。这与前一个例子颇为相像，只是这里的输出介质不依赖于在编码时间或模板例化时间对缓冲区类型的了解。图 8.4 仍然用作一个领域依赖关系图，但绑定时间已经发生了变化。表 8.3 将变成表 8.4 的样子。

所有绑定都必须是动态的，所以我们用一个完全多调度模拟法。代码可以像这样：

```
template<class CharSet>
class OutputMedium {
public:
    virtual void write(TextBuffer<CharSet> &) = 0;
private:
    virtual void encrypt(basic_string<CharSet> &s) = 0;
    . . . .
};
```

表 8.4 Buffer 差异参数的运行时处理

差异参数	意 义	领 域	绑 定	默认方法
缓冲区类型 *结构* *算法*	文本行的格式定制和处理对 Text Buffer 类型敏感	检查点、页面、版本、完整文件……	运行时间	完整文件 *虚函数*

下面是构造类 DES 的另一种方法，使用了 operator() 进行加密：

```
template<class CharSet>
class DES {
private:
    void encrypt(basic_string<CharSet> &);
public:
    void decrypt(basic_string<CharSet> &);
    void operator()(basic_string<CharSet> &s) {
        encrypt(s); }
    DES(basic_string<CharSet> key);
};

template <class CharSet>
class TextBuffer {
public:
    void write(OutputMedium<CharSet> &op) {
        op.write(*this); }
    virtual void unixWrite(OutputMedium<CharSet> &) = 0;
    virtual void databaseWrite(OutputMedium<CharSet> &)
        = 0;
    . . . .
};

template <class CharSet>
class UnixFile: public OutputMedium<CharSet> {
public:
    void write(TextBuffer<CharSet> &buf) {
        buf.unixWrite(*this);
    }
    UnixFile(const string &fileName,
        const basic_string<CharSet> key = "");
        cypher(key) { }
private:
    DES<CharSet> cypher;
    virtual void encrypt(basic_string<CharSet> &s) {
        cypher(s);
```

```
    }
    . . . .
};

template <class CharSet>
class PagedTextBuffer: public TextBuffer<CharSet> {
public:
    void unixWrite(OutputMedium<CharSet> &) { . . . . }
    void databaseWrite(OutputMedium<CharSet> &) { . . . . }
    . . . .
};

template <class CharSet>
class FullFileTextBuffer: public TextBuffer<CharSet> {
public:
    void unixWrite(OutputMedium<CharSet> &theMedium) {
        . . . .
    }
    void databaseWrite(OutputMedium<CharSet> &theMedium) {
        . . . .
    }
    . . . .
};

int main() {
    PagedTextBuffer<char> textBuffer;
    UnixFile<char> file("afile");
    textBuffer.write(file); // fully run-time bound
    . . . .
}
```

8.5 设计和结构

多范型设计产生了一个架构——组件的结构以及组件之间的关系。在差异性依赖关系图中此架构显而易见，比如图 8.2。差异性依赖关系图不是类图或者对象图，理解这一点很重要。类和对象图编码了超出架构的结构的底层的设计考虑事项，比如绑定时间。

标准 UML 记法表示的前三节中的设计的类图分别如图 8.5~图 8.7 所示。从这些图中我们可以看出：在三个图中都隐约可以看到图 8.4 的架构。差异性依赖关系图捕获了领域的深层次架构，从某种意义上来讲，它比类图更加抽象。

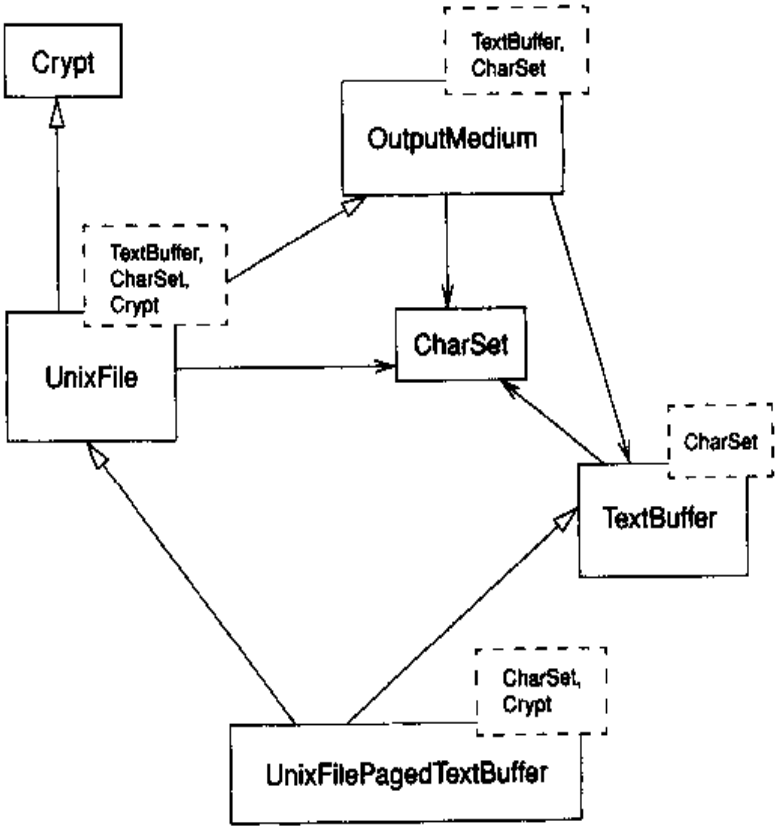


图 8.5 第 8.4.1 小节的设计的类图（第一个设计）

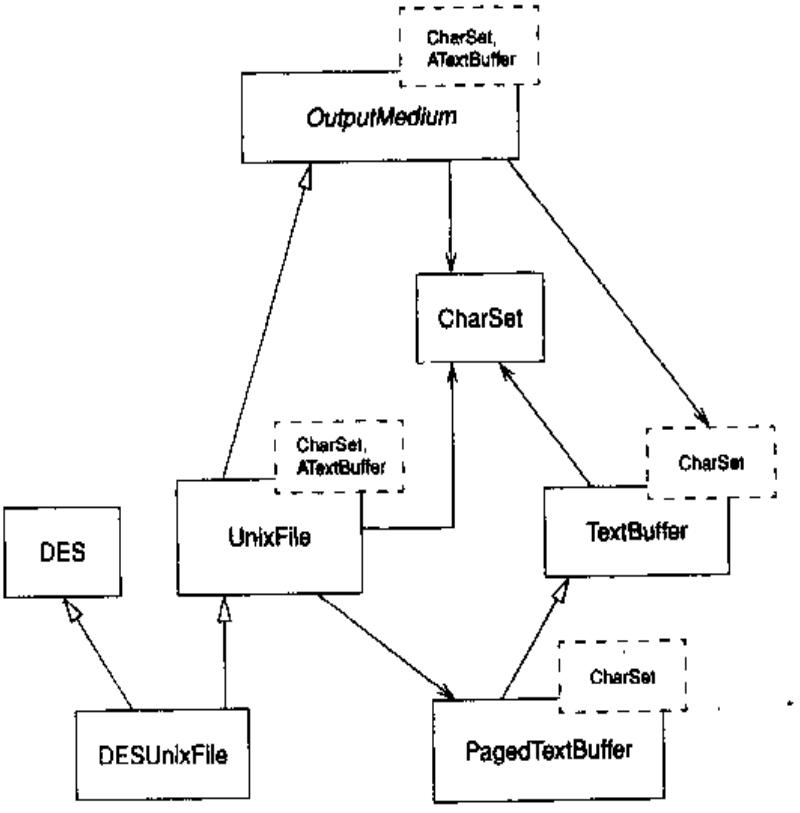


图 8.6 第 8.4.2 小节的设计的类图（第二个设计）

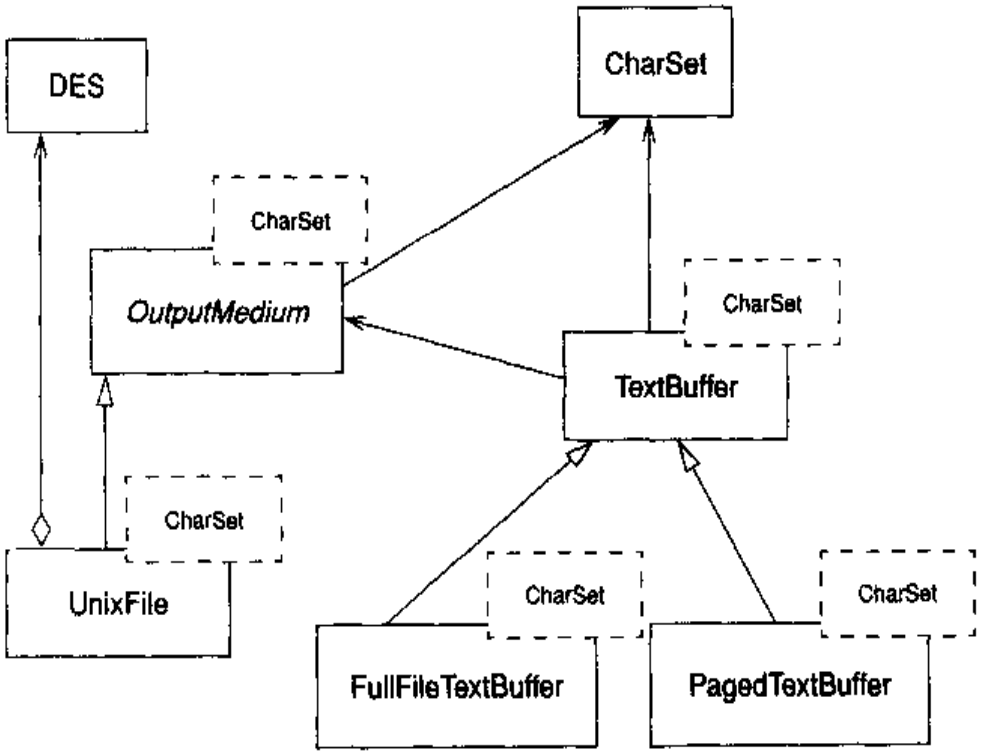


图 8.7 第 8.4.3 小节的设计的类图（第三个设计）

很明显，三个图都具有与图 8.4 相同的分类方法，但它们无法捕获到很多重要的设计维度。例如，用#ifdef 处理的差异性（比如调试代码）没有在类层次上表现出来，所以它们就没有显示在上面的类图中。虚函数和非虚函数之间的不同之处也没有显示出来，重载函数也一样。

图 8.6 与图 8.5 类似。在图 8.6 中，因为 OutputMedium 类在运行时间通过虚函数动态地绑定到一个 TextBuffer 实例，所以它没有将 TextBuffer 当成一个模板参数。

8.5.1 关于绑定时间的解释

绑定时间不同的三个设计图看似都强调了这样的设计定型，即推迟或松弛绑定“清除”一个设计定型。但类图并不总是这样。我们来考虑面向对象的设计中带有相互依赖领域的这样一个典型问题：到输出抽象族的多种形态的对象输出。

我们假设正在设计一个气候分析和显示系统，其中“云”是主要的抽象，包括卷云、积云、雨云、漏斗云和一些其他类型的云：

```
class Cloud {
public:
    . . . .
};
```

```
class Cirrus: public Cloud {
    . . . .
};

class Tornado: public Cloud {
    . . . .
};
```

我们可以有多种不同类型的输出格式化程序，包括三维 Phong 阴影绘图程序（可以绘出美丽的云图）、等压线和等温线显示程序（可以绘出常压或云内温度的等高线）和风向显示程序（可以绘出表示风速的小箭头）：

```
class Display {
public:
    . . . .
};

class Phong3D: public Display {
    . . . .
};

class IsoBar: public Display {
    . . . .
};
```

我们可以将云族和输出类型族都作为领域来看待。很快就可以发现这两个领域是相互依赖的。每个领域的输出格式化程序都知道如何为其介质绘制特定种类的云，但它需要先知道它所绘制的每个对象的类型。每一种云的接口（比如绘图函数）都依赖于显示设备。在最不利的情况下，这些函数中有些不是类函数，但它们可能用于特定类型的输出设备上的特定类型的云（比如，当为绘图程序使用 Web 浏览器时，就可以使用一个内建的 Java 小程序获得一个动画的漏斗云。）

假设我们可以使用编译时的绑定来解决成员函数（比如 draw）的选择问题。相应的 C++解决方案比较琐碎：

```
void draw(const Tornado &t, IsoBar &d) { . . . . }
void draw(const Tornado &t, IsoTherm &d) { . . . . }
void draw(const Cumulus &fluffy, Phong3d &d) { . . . . }
void draw(const Cirrus &c, Phong3D &d) { . . . . }
. . . .
```

列表有些长，并且友元关系非常凌乱（每个类必须为访问它的每个函数声明友元关

系）但设计很直接。

现在，假设依赖性必须是运行时绑定。也就是说，只能有一个函数：

```
void draw(const Cloud &, const Display &);
```

此函数可以接受任何 Cloud 和任何 Display，并执行适当的任务。这是一种要在C++中实现的、具有挑战性的设计方案，通常要使用类型字段来模拟多重调度（参见[Coplien1992]，第 9.7 节）。在这种情况下，推迟的绑定大大增加了实现的复杂性，而不是简化实现。

在 CLOS 中，一种有与 C++重载类似的语法的语言结构拥有运行时绑定的语义，并完全使用多重调度。为特定共同性分析选择一种适当的设计技术要依赖于所选择的实现语言。

8.6 另一例子：有限状态机

在这个例子中，我们将说明：即使对低层次的设计，也可以使用多范型设计。有限状态机（FSM）通常被当成一个可以用单个类来表达的低层次的抽象。但如果要将一个通用 FSM 放在广泛应用的库中，我们就要找到库的很多用户的差异参数，以便每个用户都可以调整设计以满足具体的需要。

我们依据支持一般实现和外部接口的一些抽象来审视 FSM。FSM 是一个状态集，是值的一个有限集合，其中的每个值都表示 FSM 所代表的系统的一种有关配置。状态变量通常是整型和枚举类型，但它必须是一个字符串或者任意的类类型。这将是最重要的一个差异参数。这些子领域都在领域词典（参见第 2.2.2 小节中关于 FSM 领域词典的论述）中有所预示。

FSM 会响应“消息”而改变状态。通常，我们将每条“消息”实现为一个成员函数。向 FSM“发送一条消息”就意味着调用了一个成员函数。当然，我们也可以用单个函数（此函数应当可以接受更常用的消息作为数据结构参数和枚举消息类型）来表示这个接口。我们想要让所设计的实现能够支持其中任意一种类型。

FSM 也执行动作产生输出。在完全通用（Mealy）的 FSM 中，所执行的动作是当前状态和输入消息的一个函数。（在 Moore 机中，输出只依赖于当前状态，这里的讨论描述的是 Mealy 机）程序员编写这些动作来执行系统的任务——这是真正的目标所在。这些动作放在一起就是 FSM 的一个重要的差异参数。

所有这些差异是相互独立的，FSM 中有两个共同性领域。第一，所有的 FSM 共

享同种接口。所有 FSM 的领域（称为 **AbstractFSM**）定义了一个普通接口。差异参数包含自定义转换类型，以及针对 **State** 和 **Stimulus** 的类型。我们在表 8.5 中捕获这些领域。

表 8.5 共同性领域（**AbstractFSM**）的 FSM 转变分析（共同性：结构和行为）

差异参数	意 义	领 域	绑 定	默认值
用户机（User machine） *结构* *算法*	所有通用的FSM都知道如何在任意的UserFSM中添加转换	成员函数可接受激励参数的任意类（参见最后一行）	编译时间	无 *模板*
状态（state） *类型*	如何表示FSM的状态	任何离散类型	编译时间	无 *模板*
激励（stimulus） *算法*	使机器在状态之间进行转换的消息的类型	任何离散类型	编译时间	无 *模板*

如果绑定 **AbstractFSM** 的所有差异参数，我们就可以生成一个高层次抽象的族成员，它表示响应给定激励并具有给定状态范围的所有可能状态机。此时，**AbstractFSM** 领域提供了支持自定义 FSM 的签名和协议的族。这并不是一个无正当理由的设计决定。我们可以在 **AbstractFSM** 层次上定义带有对多个类似的 FSM 的多种形式的接口的函数。这些函数可交替处理多个 FSM。

第二，所有的 FSM 都共享同种实现。我们在一个 FSM 的实现中找到当前状态变量和一个状态映射表。整个结构对所有差异是共有的，其差异参数与 **AbstractFSM** 的差异参数类似。这本身是一个有趣的子领域，我们称之为 **ImplementationFSM**。表 8.6 捕获了此子领域的差异性分析。

还有第三个领域，即用户指定的机器本身。此子领域捕获了很多差异性，共同性被推入到另外两个子领域。我们称这个领域为 **UserFSM** 领域。它从 **AbstractFSM** 子领域的一个族成员中获得其协议接口，从而使得它依赖于 **AbstractFSM** 子领域，而 **AbstractFSM** 反过来又依赖于 **UserFSM** 领域——这是一种循环依赖性（图 8.8）。由于状态转换定义和动作语义的实现的原因，**ImplementationFSM** 子领域反过来又依赖于 **UserFSM** 子领域。**UserFSM** 差异性表可在表 8.7 中找到。

表 8.6 共同性领域（Implementation）时 FSM 转变分析（共同性：结构和行为）

差异参数	意　义	领　域	绑　定	默认值
UserFSM *结构* *算法*	为了实现状态/动作映射，实现必须知道用户自定义的动作和转换的类型	参见表 8.5	编译时间	无 *模板*
状态（state） *类型*	如何表示 FSM 的状态	任何离散类型	编译时间	无 *模板*
激励（stimulus） *算法*	使机器在状态之间进行转换的消息的类型	任何离散类型	编译时间	无 *模板*

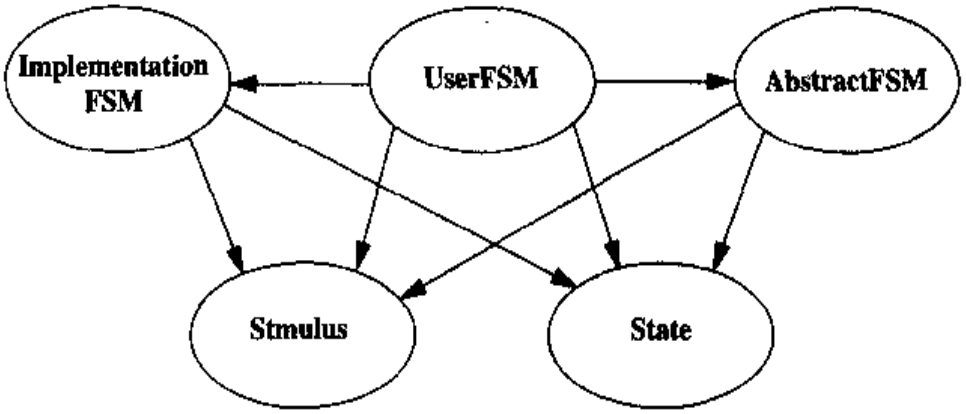


图 8.8 FSM 示例的领域图

表 8.7 共同性领域（UserFSM）的 FSM 转变分析（共同性：总的行为）

差异参数	意　义	领　域	绑　定	默认值
AbstractFSM *结构* *算法*	UserFSM 使用来自 AbstractFSM 领域的某些族成员的协议	参见图 8.5	编译时间	无 *继承*
状态（state） *类型*	如何表示 FSM 状态	任何离散类型	编译时间	无 *手动编码和 typedef*
激励（stimulus） *算法*	使机器在状态之间进行转换的消息的类型	任何离散类型	编译时间	无 *手动编码和 typedef*
动作（action） *算法*	每个 UserFSM 都在转换函数中实现自身的语义	映射一个 Stimulus 参数的任意多个函数和到一个状态的当前状态	编译时间	无 *继承*

每个 FSM 都是惟一的，所以差异性可以进行手动编码。这种惟一性归功于设计解析，正是设计解析将共同性推入到 **ImplementationFSM** 和 **AbstractFSM** 领域中。为方便抽象，反复出现的参数（比如 **Stimulus** 和 **State**）的声明可以用 typedef 来表达。

注意：这个设计还展示了相互依赖的领域，即 **AbstractFSM** 和 **UserFSM** 相互依赖，如图 8.8 所示。

我们用一个类 AbstractFSM 着手启动实现，这个类捕获了 FSM 整个领域的语义。共享相同协议的 FSM 所定义的子领域表示在下面的 ImplementationFSM 类（直接从 AbstractFSM 派生而来）中：

```
template <class M, class State, class Stimulus>
class AbstractFSM {
public:
    virtual void addState(State) = 0;
    virtual void addTransition(Stimulus, State, State,
        void (M::*)(Stimulus));
};
```

仅仅通过忽略来自基类的转换函数，并将它们添加到针对每个机器的派生类中，我们就可以在 **UserFSM** 中捕获这些转换函数的差异性：

```
class UserFSM: public AbstractFSM<UserFSM, char, char> {
public:
    void x1(char);
    void x2(char);
    void init() {
        addState(1);
        addState(2);
        addTransition(EOF,1,2,&UserFSM::x1);
        . . . .
    }
};
```

注意 Barton 和 Nackman[Barton+1994]将派生类型作为参数传递给生成基类的模板的方法。

类 ImplementationFSM 捕获了 **GenericFSM** 领域的语义。这也是我们隐藏跨所有 FSM 的共同性的地方：内部映射表（使用一个称为 map 的通用库抽象，它实际上是一个相关阵列）、当前状态和状态的数量。它还捕获了所有 FSM 的共同行为：构造、添加状态、添加状态间的转换和 fire 函数（它让机器在状态之间循环转换）：

```
template <class UserMachine, class State, class Stimulus>
class ImplementationFSM: public UserMachine {
public:
    ImplementationFSM() { init(); }
    virtual void addState(State);
    virtual void addTransition(Stimulus, State from,
        State to,
        void (UserMachine::*)(Stimulus));
    virtual void fire(Stimulus);
private:
    unsigned nstates;
    State *states, currentState;
    map<Stimulus, void(UserMachine::*)(Stimulus)>
        *transitionMap;
};
```

通过使用继承（ImplementationFSM: public UserMachine），此声明捕获了**ImplementationFSM** 领域对 **UserFSM** 领域的依赖性。现在，从下面这些声明来构建 ImplementationFSM 模板就比较繁琐了：

```
ImplementationFSM<UserFSM, char, char> myMachine;
. . . .
```

这个声明表明：设计已经充分地把用户和共同性屏蔽开来，同时表达了模板参数中和 UserFSM 类中的函数的差异性。此代码生成了一个相当精细的设计，比我们从一个简单的 FSM 或从一个与此解决方案所进行的同等深度的面向对象的分析中预想的更加复杂。然而，连接到库的用户接口却比较清晰和简单，非常适当地表达了所需要的设计差异，如图 8.9 所示。

此图充分扩展了 UML 标记。ImplementationFSM 和 AbstractFSM 是模板。类

```
AbstractFSM< UserFSM, char, char >
```

和

```
ImplementationFSM< UserFSM, char, char >
```

是这两个模板的例化，此时 UserFSM（以及 State 和 Stimulus，如图中的 char 参数所示）被作为差异参数。类 UserFSM 将 AbstractFSM 例化作为基类，同时又作为 ImplementationFSM 例化的一个基类。UserFSM 对其接口和实现同时使用了它的两个相邻的类。

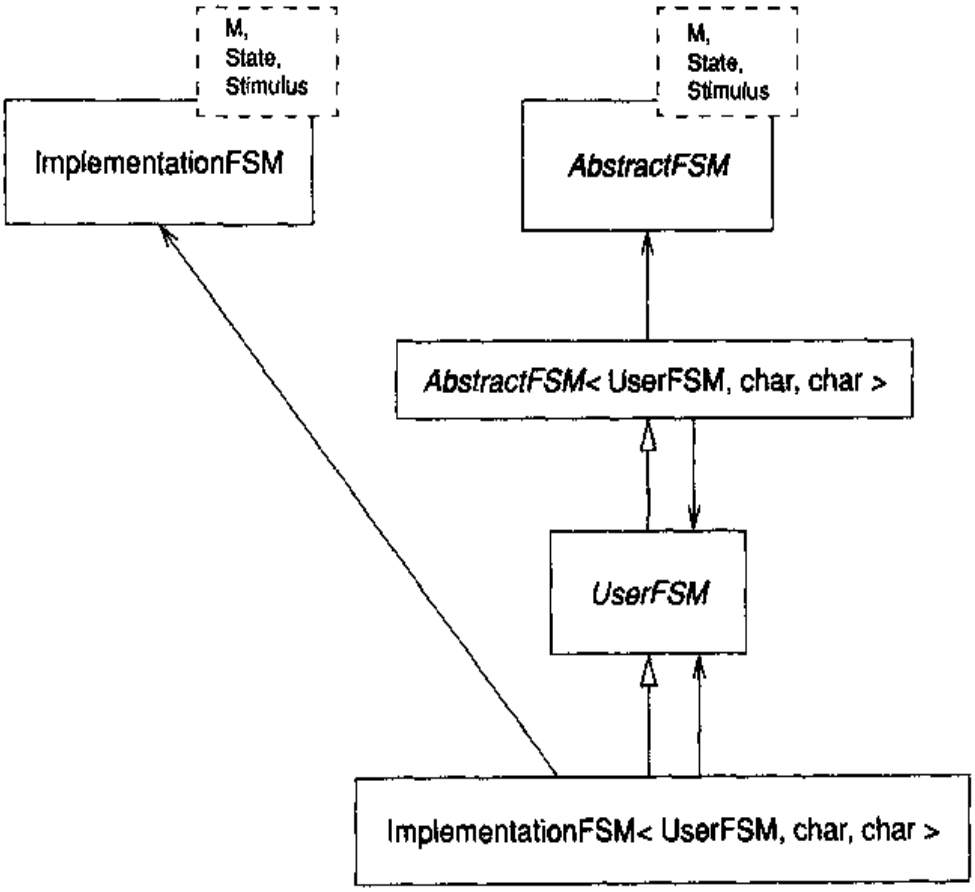


图 8.9 解决方案的类图

这并不常用，但确实是一种很有效的设计。这里的表示法省去了使其成为一个“工业级别”的 FSM 的很多细节。但它仍然是一个小到足可以理解，大到足可以延伸想象的例子。这个例子表明：即使在设计的最底层，设计者仍然可以使用多范型设计。或者可能不是一个如此低的层次，因为这些设计决定中有很多与系统的基于状态机范型的基本结构有关。

“状态机范型”是一个惹眼的措辞。一个状态机本身就是一种范型，如果是这样，我们可否使用 C++和多范型设计来捕获其共同性维度？我认为状态机是一种范型[①]，但除最简单的状态机以外，我不认为 C++很“自然地”捕获了其重要的差异参数。有些语言的目标就是可以更自然地捕获状态机语义，SDL[Turner1993]就是这样的一个例子。很多项目可以构建适合它们的领域的自身的状态机语言。设计者面对这样一个设计决定：是使用这里状态机架构，还是使用一个针对状态机进行调谐的 AOL？第 7.2 节探究了在此设计决定中所进行的权衡。

① 有资料表明，我的同仁 Tim Budd（也是多范型设计中的早期实践者[Budd1995]）在这一点上与我的意见并不一致。

8.7 基于模式的方案策略

设计模式可用作一种打破循环依赖性的高级技术。例如，我们来考虑一个行为（比如 write），其实现依赖于两个不同的抽象（比如 **File** 类型和 **Text Buffer** 类型）。我们可以将相互依赖的行为对象化，并将其放到一个 **Strategy** 对象中；而不是将行为隐藏在 **File** 或 **Text Buffer** 中——这将导致一个直接违反另一个的封装。如果在运行时间行为同时依赖于 **File** 和 **Text Buffer** 类型，那么就可以将其当作多重调度问题来对待，使用多重调度代码模式[Coplien1992]或 **Visitor** 模式[Gamma1995]。**Strategy** 模式也可以很好地实现不同形式的加密。

[Gamma1995]深入地介绍了模式解决方案，而本书的第 9 章也探究了多范型设计到模式的关系。多范型设计常可以提供“特定共同性、差异性、绑定时间和例化应当使用哪种模式”的线索。其他问题则需要超出常规共同性和差异性分析的模式解决方案，请参见[Coplien1996a]中的论述。

8.8 小结

本章将多范型设计带入到一种极端的情况：捕获多种范型之间的相互作用。多范型设计可以帮助设计者找到适用于给定领域的范型，同时使设计者可以随意使用各人的或项目范围内的风格和规约来编织这些范型。多范型设计也可以识别“独立领域”之间的递归设计依赖性。这种设计比较难以规范化，要依靠实现技术和应用领域的经验。

到此为止，本书已经着重讨论了 C++可以自然地表达的抽象技术。有时，从更高层次的角度来审视问题，往往可以受到启发，从而获得一个更强大的系统划分。这些划分有时通过间接的方法（常常不服从于系统化方法甚至直觉）来解决问题。设计者已经捕获了这些在历史上很成功的划分作为软件模式。这些模式中有一些超出了多范型设计，但有些与本书设计的内容紧密相关。第 9 章探究了多范型设计和设计模式的新出现的规程之间的关系。

第 9 章
用模式扩充方案领域

本章将模式引入到方案领域词汇表中。模式扩展了 C++，以表达 C++结构无法直接表达的共同性与差异性的丰富组合。模式通常用于解决消极差异性所带来的问题。

9.1 代码模式与模式的价值

值得信赖的方法论者不能避开模式来讨论软件设计。模式为有经验的程序员所熟知的技术和成熟的、经过证明的设计结构提供了一个词汇表。这些技术中有些首先被捕获为“代码模式”（idiom）[①][Coplien1992]，并且已经在 Gamma 等人的著作中具体化了[Gamma1995]。

软件模式解决了常见的软件问题。对正式的设计方法来说，模式是一个有用的辅助工具，它填补了某些方法留下的缺陷。没有哪种常规的设计方法会直接带来《Design Pattern》一书[Gamma1995]中所描述的所有模式。它们提供了供有经验的设计者使用的关键结构，以补充面向对象的设计方法所生成的结构。这些结构中的一部分补充了多范型设计所创建的结构，一部分与多范型设计生成的结构紧密相关。本章探究设计模式与多范型设计的交集部分。

模式不是多范型设计的相关附属物，它本质上与共同性和差异性分析相联系。模式本身是很多程序所共有的常见设计单位，所以可以将它们添加到我们认为会在方案领域中发现的共同性里。很多模式没有将差异性归到经常性出现的设计结构中，以适应频繁

① Andrew Koenig 称它们为 cliches。idiom 和 cliche 是一回事，但 Koenig 认为，术语 idiom 比 cliche 更正式、更常用。

出现的变化。例如，[Gamma1995]的 Bridge 模式将正在改变的表示方法从公用接口中划分出来。Gamma 等人最早在《Design Patterns》一书[Gamma1995，第 30 页]中用设计差异性将设计模式制成表格，本章中我们将经常回到这个表上来。Wolfgang Pree 将这些变化轨迹称为“热点”[Pree1995]，它类似于多范型设计中的差异参数。

9.1.1 语言之外的模式

模式通常表达了编程语言无法直接表达的内容。鉴于多范型设计只在语言特性层次上构建解决方案，模式更加清楚地表明了如何解决问题，并具有丰富的设计知识。好的模式可以知道读者做什么、怎样做、以及结果是什么。模式有多种形式，但所有的模式都描述了一个问题和一个解决方案。大部分好的模式都讨论了通常称为“人工转移”（force）的设计折衷。一种模式应当讨论它是如何转换一个设计的，应当考虑什么样的补充模式，以及模式的优点和可靠性。模式是比简单的语言技术更丰富的解决方案，我们应尽量利用模式来解决设计问题。

很多模式都捕获共同性，并参数化差异性，但我们不应将所有的模式都解释为共同性和（或）差异性编码。一种模式就是某种环境下某个问题的文档化解决方案。我们没有理由相信通过这样一种宽泛的定义，任何模式都可以适用于多范型设计的任何共同性类别。模式是更加通用的。例如，**Leaky Bucket Counter**[PloP1996]模式介绍了如何管理阶段性减少的错误计数器，以补偿错误计数增量。此计数器只在错误频率非常高的时候才达到极限值。我们当然可以在 C++中实现这种模式，但那不是一种真正的结构模式。况且，就共同性和差异性来讨论它是比较困难的（除非在漏斗计数器自身的领域中，泄漏率和频率成为了设计的“热点”）。

更一般地来讲，多范型设计易于同软件抽象区段中的模式交叠在一起。我们可以临时将模式分成三种：框架模式、设计模式和代码模式。框架模式（比如 Client/Server）通常与编程语言因素关系不大。与之相对，代码模式与设计者在实现中必须要面对的编程语言结构紧密联系在一起。设计模式介于二者之间，它将应用结构与方案结构结合起来——这与多范型设计所描述的是同一种转变。

9.1.2 模式和多范型设计

既然模式解决了这些问题，为什么还需要多范型设计？模式解决了某个领域（应用领域或者方案领域）所特有的问题。而另一方面，多范型设计没有依靠以前的解决方案，

而且它并不关心应用领域。大多数模式没有处理这些领域之间的映像[①]。多范型设计分析相互依存的应用领域和方案领域，从而得出一种折衷平衡的结构。模式提供了大量（但数量固定）的解决方案，我们可以对这些方案进行自由组合，而多范型设计是根据有限的一组编程语言结构来构建其解决方案的。从某种程度上来说，多范型设计可以比任何模式集合更通用。要想获得与多范型设计相同的通用性，模式的分类必须非常大，但这样又难以搜索。

如果可以识别某种具体的模式能够很好地处理的共同性和差异性，设计就会从模式说明中的支持资料中获益：基本原理、推荐的实现结构、替代解决方案、性能区分和其他折衷方案。正如早期设计中我们用共同性分析来调整第 5 章的一个示例中的面向对象的使用一样，这里，我们用共同性分析来指向通用模式。模式说明提供了更特化的设计和实现细节方面的知识。

那么，如何进行相互关联的模式设计和多范型设计？我们可以从模式使用的三个方面来理清它们的关系：

1）作为方案领域分析中出现的共同性/差异性的别名（第 9.1.3 小节）。

2）作为比它们下面的 C++结构更加抽象的抽象（第 9.1.4 小节）。

3）作为处理消极差异性的强有力机制（第 9.1.5 小节）。

除这些关系之外，模式还涉及很多与多范型关系不大的设计结构。这里我们不再探究这样的模式。更广泛一点来看，模式对多范型设计进行了补充。本章将只讨论这两种设计技术的交集部分。

9.1.3 方案领域结构的别名

共同性和差异性是大部分设计方法和工具的主要内容。很多常见的共同性和差异性的模式都被做成编程语言，以便支持我们称之为范型的设计风格。共同性和差异性的某些组合并不是很通用，还不足以将其做成一种编程语言，但在我们的设计词汇表中它们是足够通用的。这些结构就成为补充了通用应用领域和方案领域的主要抽象的模式。

Template Method[Gamma1995]就是这样一种模式。我们在私有虚函数代码模式中（[Coplien1992]，第 11.1 节）和其他早期的 C++实践中找到这种模式的根源。模板方法是这样的一种算法：它拥有整体上共同的结构，具体步骤视具体应用而有所不同。大部分编程语言都无法轻易地表达算法中细微的运行时差异（尤其是比一个函数还要小的差异）。模板方法按照更细微的算法（相对于主要算法而言，可以动态地进行调度）来表达

① 从技术上来讲存在一些例外的情况，比如 Norm Kerth（[PLoP1995]的第 16 章）的 Caterpillar's Fate 模式语言。但 Norm 的著作与这里的多范型设计在焦点和层次上是不同的。

主要的算法。主要的算法对共同性进行编码。算法中的差异性可以通过虚函数（大部分面向对象的编程语言都可以很好地支持它）归到继承层次结构中。

Pree[Pree1995]显然是将 **Template Method** 当作共同性和差异性分析的一个用例。他用“热点”这个术语来描述差异的位置，比如 **Template Method** 的算法差异，我们要在一个健壮的设计中对其进行参数化。Pree 的 Hook 模式与 **Template Method** 模式紧密相关。这两种模式都捕获算法结构作为一个共同性和差异性类别。

并非共同性和差异性的所有组合都需要别名。以同种方式重载继承层次结构中每个类中的同一个函数，这种技术可以取什么样的别名？如果枚举出所有这些组合，就生成了一个百科辞典式的词汇表。而且，从多范型设计中使用的术语的意义上说，并非所有的模式都建立在“共同性”和“差异性”上。

9.1.4 比编程语言结构更高级的层次

我们来考虑 **Iterator** 模式[Gamma1995]。这是又一个捕获设计共同性和差异性的模式。与前一节介绍的模式的共同点是都需要对一个集合进行迭代，而不管它是如何表示的。不同点是集合结构的实现和集合所包含的元素的类型。大多数面向对象的编程语言都有聚合类型，且迭代程序是相同的，并以一种简便的方式同聚合进行交互。有一些语言（比如 Lisp）拥有内建迭代结构，但大部分语言都使用一个共同的模式将迭代构建到库抽象中。

在大多数情况下，我们将列表迭代当作一种比具体编程语言的语法中所表达的概念更加通用的概念。模式为迭代程序的语义和其他高级的设计结构（这种结构必须足够抽象，可以跨编程语言进行端口通信）提供了一种“标准”。

注意，这些“高级”模式中有很多都是这样一种结构的别名：此结构是某种编程语言文化的一部分，但又不是语言自身的一部分，正如前一节所介绍的那样。

9.1.5 消极差异性

第 5.1.4 小节讨论了带来一类特殊的消极差异性的共同性/差异性对。很多类可能共享共同的行为，但它们可以有不同的结构。下面这段代码中显示了这样一个具体的例子：

```
class Complex { // size = 16 bytes
private:
    double realPart, imaginaryPart; // 16 bytes of space
public:
    Complex &operator+=(const Complex &);
```

```
    friend Complex operator+(const Complex &, const
        Complex &);
    . . . .
};

class Real: public Complex { // size should be 8 bytes
private:                     // (presuming implementation
                             //  as a double)
    // ??? add a real part and waste base class data?
    // make base class data protected, and waste
    // imaginaryPart?
    . . . .
};
```

不但 Real 和 Complex 的实现不同，而且根据 Complex 来构建 Real 的实现也不可能。尽管如此，这两个类的外部行为相同，并遵循所有关于子类型导出的合理法则。C++语言对二者都使用同一种语言结构——继承，这里的一致性并不能满足我们的需要。这表现为多范型设计中的消极差异性，因为派生类 Real 必然违背显然更加抽象的基类 Complex。回顾表 6.2，我们看到多范型设计没有提供 C++语言方案来解决这种配对。所以，我们回到模式上来。

一种解决方案是将实现和接口打破成为与指针联系在一起的独立继承：[Gamma1995] 的 **Brigde** 模式。此方案可以像下面这样：

```
Complex operator+(const Complex &n1, const Complex &n2) {
    // machinery to orchestrate multiple dispatch
    // see [Coplien1992], sec. 9.7
    . . . .
}

class NumberRep { };

class ImaginaryRep: public NumberRep {
friend class Complex;
friend class Imaginary;
    double imaginaryPart;
    ImaginaryRep &operator+=(const ImaginaryRep&);
    . . . .
};

class ComplexRep: public NumberRep {
friend class Complex
    double realPart;
```

```
    ComplexRep &operator+=(const ComplexRep&);
    . . . .
};

class Complex {
public:
    Complex(): rep(new ComplexRep()) { . . . . }
    ~Complex() { delete rep; rep = 0; }
    Complex &operator+=(const Complex &c) {
        (static_cast<ComplexRep *>
            rep)->operator+=(c.rep);
        return *this;
    }
    . . . .
protected:
    Complex(NumberRep *);     // for use by derived
                              // classes
    NumberRep *rep;           // the Bridge pointer
};
class Imaginary: public Complex {
public:
    Imaginary(): Complex(new ImaginaryRep()) {
        . . . .
    }
    ~Imaginary() { }
    . . . .
};
```

这解决了第 6.7 节中所提出的问题，并阐明了图 6.4 的解决方案结构。另请参阅第 6.11.1 小节“数据取消的一个例子”中的一个类似的例子。

很明显，模式解决方案与多范型设计是相关的——至少它们在某些情况下解决了类似的问题。实际上，有些设计模式可以更加正式地作为共同性和差异性的特殊配置与多范型设计产生联系。换句话说，我们可以使用多范型设计来使常用的设计模式子集规范化。

9.2 常用模式中的共同性和差异性

第 6 章将 C++方案领域结构制成了表 6.2。现在，我们将“模式方案领域”添加到 C++方案领域中，它可以代表共同性和差异性的新的、丰富的组合，以支持多范型设计的后端。请参见表 9.1 的总结。此表可以当成是对 C++方案领域表的一种扩展，所以在结合应用领域结构和方案结构的时候，需要同时考虑这两个表。

表 9.1 用模式处理共同性和积极差异性

共同性	差异性	绑 定	例 化	模 式
函数名和语义	细微算法	运行时间	N/A	Template Method（第 9.2.3 小节）
	算法	运行时间和编译时间（默认）	N/A	Unification（第 9.2.6 小节） +Template Method
	算法：差异的参数是一些状态	运行时间	是	State (第 9.2.5 小节)
有关运算和部分结构（积极差异性）	总算法	运行时间	N/A	Strategy（第 9.2.4 小节） Singleton（第 9.2.7 小节）
	状态值	源码时间	一次	Strategy（使用模板，第 9.2.4 小节）或 Unification（第 9.2.6 小节）
	总算法	源码时间（或编译时间）	N/A	
有关运算，非结构	不兼容数据结构	任意	是	Bridge 或 Envelope/Letter（第 9.2.2 小节）

9.2.1 领域工程技术之外的模式

当然，表 9.1 没有对所有有用的模式进行编码。很多模式超出了多范型设计的共同性和差异性结构，来处理从实现效率（比如在 Flyweight 模式中的效率）到一般系统层次架构（比如 Client/Server 模式）的问题。正如可用作任何设计方法的辅助设备一样，这些模式可以用作多范型设计的辅助设备。甚至[Gamma1995]的设计模式也可以当成是面向对象的方法的辅助设备。

接下来的几节将对这些模式以及它们的共同性、差异性和实现考虑事项进行更加详细地讲解。

9.2.2 Bridge 模式

图 9.1 的 **Bridge** 模式打破了“抽象及其实现之间的绑定”（[Gamma1995]，第 153 页）。它通过删除表达抽象的类的细节，并通过指针打破客户端可见抽象中的编辑依赖性，从而对客户端隐藏了实现。这也称为 handle/body idiom[Coplien1992]。《Design Patterns》一书[Gamma1995]指出 **Bridge** 所捕获的差异性是“一个对象的实现”，但没有将整本书的模

式显式地制成表格。

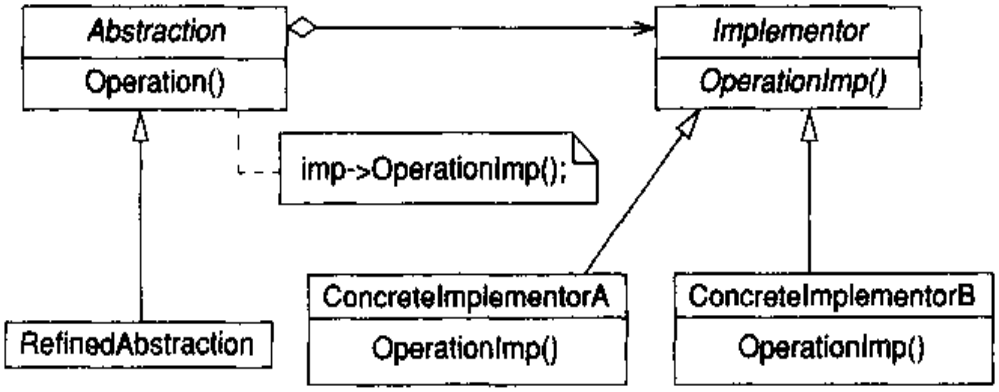


图 9.1 **Bridge** 模式的结构（改编自[Gamma1995]，第 153 页）

考虑 **Bridge** 的方式之一是假设跨族成员的结构不存在共同性。每个族成员都重新产生其自身的结构（即不是构建在任何基础之上）。这就意味着族成员可以展示完全不相关的结构和不需要担心消极差异性的问题的结构。共同性来自于族成员（基类接口）的相关签名的更广阔语境。

9.2.3 Template Method 模式

Template Method（图 9.2；[Gamma1995]，第 325 页）捕获了算法族的主要共同性。它突出了跨应用不同的算法“片断”，并将这些差异放到一个派生类中。Gamma 等人[Gamma1995]告诉我们：**Template Method** 捕获了“算法的步骤”作为一个差异性。共同性是相关的算法框架。

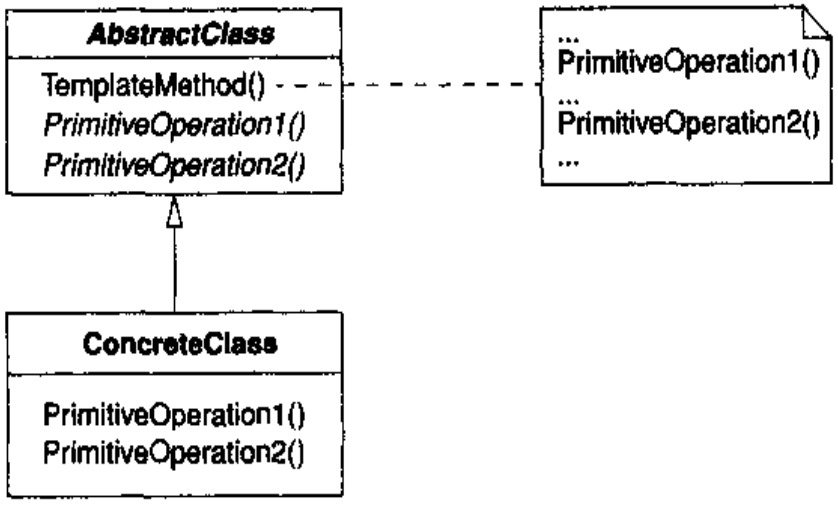


图 9.2 **Template Method** 的结构（改编自[Gamma1995]，第 327 页）

9.2.4 Strategy 模式

Strategy（[Gamma1995]，第 315 页）与 **Template Method** 类似，只是差异性的粒度不同。

Template Method 处理“片断”，而 **Strategy** 则假设算法间没有共同性的结构（在表 9.1 中比较这两种模式的“差异性”条目）。在《design patterns》一书[Gamma1995]中，**Strategy** 的差异性是“一种算法”。惟一的共同性是算法族与它们的客户端之间的合同。《design patterns》为 **Strategy** 推荐了两种实现。尽管这两种实现都目标相同，但因为它们的绑定时间不同，所以多范型设计将它们区别对待。这就是表 9.1 中两次出现 **Strategy** 的原因。

Strategy 的第一种实现是 **Template Method** 的变化形式，每个类只带有一个函数，如图 9.3 所示。虚函数方便了运行时绑定，并将其集成到 C++编程语言及其类型系统中。**Strategy** 更进一步，将差异性推到了运行时间。

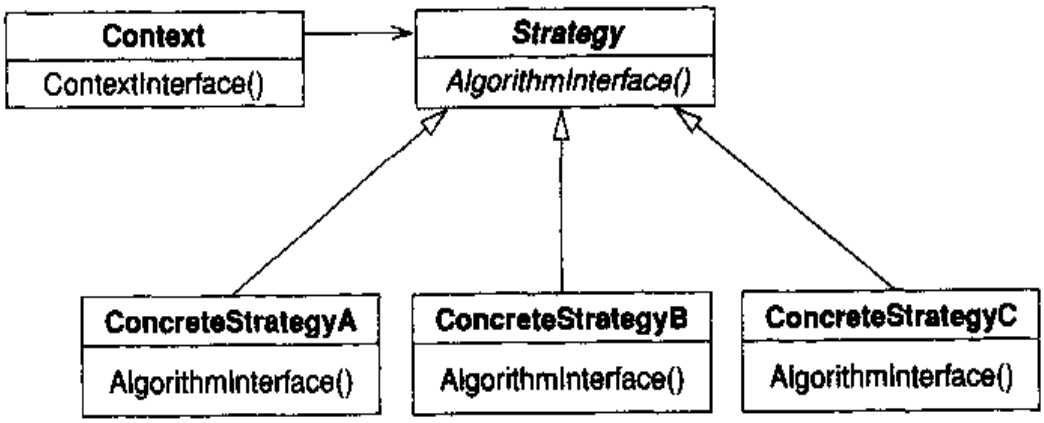


图 9.3 **Strategy** 的结构：第一种实现（改编自[Gamma1995]，第 315-316 页）

在第二种实现中（参见[Gamma1995]，第 319 页），我们在编译时间使用模板将 **Strategy** 绑定到客户端。代码类似下面这样：

```
class MyEncryptionAlgorithm {
public:
    string crypt(const string &);
};

template <class AnEncryptionAlgorithm> class TextBuffer {
    /* static */ AnEncryptionAlgorithm encryptor;
public:
    void someMemberFunction() {
        string c, s;
```

```
        . . . .
            c = encryptor.crypt(s);
        . . . .
    }
};

TextBuffer<MyEncryptionAlgorithm> aBuffer;
```

或者，我们可以别出心裁一些，使用函数：

```
class MyEncryptionAlgorithm {
public:
    string crypt(const string &);
    string operator()(const string &s) { return crypt(s); }
};

template <class AnEncryptionAlgorithm> class TextBuffer {
    /* static */ AnEncryptionAlgorithm encryptor;
public:
    void someMemberFunction() {
        string c, s;
        . . . .
        c = encryptor(s);
        . . . .
    }
};

TextBuffer<MyEncryptionAlgorithm> aBuffer;
```

注意：C++重载是 **Strategy** 的一个特殊的或退化版本，其中的差异参数（退化）为函数签名中的类型。在模板和虚函数的例子中，差异参数在应用领域中（是算法本身）。

这与C++方案领域分析表（表6.2，该表规定了相关运算的共同性、算法中的差异性、编译时绑定、可选例化的继承）中的条目类似。绑定上存在细微的不同。如果使用了 **Strategy** 的模板实现，差异参数在模板例化时就是显式而明确的。如果使用继承，则差异性必须在类名中进行编码。实现模板版本的这种代码是紧凑而富于表达的。继承技术为基类与差异参数的每次结合都生成了一个新的派生类——这是管理疏漏，影响了全局名字空间。

回到第3章的运行文本编辑器的例子，两个声明

```
TextBuffer<DESEncryption> myTextBuffer;
TextBuffer<NoEncryption> tempFileBuffer;
```

都使用了同一个源抽象 TextBuffer。将它们与下面的声明进行比较：

```
TextBufferForDESEncryption myTextBuffer;
TextBufferWithoutEncryption tempFileBuffer;
```

下面的类声明留给读者自己思考。模板版本比较接近于源码时绑定，继承版本容易让人联想到运行时绑定（参见第 3.5.4 小节）。

粒度上也存在不同。程序员可以使用继承方便地表达几种算法是如何相对于一个基类的共同性而不同的。我们可以用 **Strategy** 实现这一目标，但继承更富于表现，也更加方便。这两个因素（简便性和粒度）是区分这两种技术的语用理论。

9.2.5 State 模式

State 模式的实现构建在与之存在细微差异的 **Strategy** 的基础上。Context 对象的状态中的变化导致其选择一个不同的 **Strategy** 对象。**State** 和 **Strategy** 在目的上有所不同。如果用 **State**，算法中的变化紧跟着对象状态的变化。**State** 的差异参数是 Context 对象的状态。**Strategy** 的第一种实现几乎与 **State** 没有区别。**Strategy** 的第二种实现将函数本身当作差异参数。**State** 支持比 **Strategy** 的第二种实现更迟的绑定：行为是设计用来在对象生命期内改变 **State** 的，而在例化时绑定 **Strategy** 行为更加常用。

9.2.6 Unification 模式

Pree 的 **Unification** 模式建议用在普通继承和虚函数上放一个有趣的 spin 的实现来集合 **Strategy** 模式的两种实现。我们从 **Template Method** 模式开始，根据领域分析的指示，此分析建议使用带有细微差异的大型的算法共同性：

```
class Base {
public:
    void commonCodeFunction() {
        . . . .
        this->hookFunction();
        . . . .
    }
private:
    virtual void hookFunction() = 0;
};
```

现在假设相同的领域分析建议对对应于 hookFunction 的差异参数使用主要的或默认的算法。我们可以在应用领域分析表的“默认值”栏中找到这种算法。

```
class Base {
public:
    void commonCodeFunction() {
```

```
        . . . .
        this->hookFunction();
        . . . .
    }
private:
    virtual void hookFunction() {
        . . . .
        // code for the default case
        . . . .
    }
};

class Derived: public Base {
    void hookFunction() {          // specialized hook
                                   // function
        . . . .
    }
};

Base commonCase;
Derived specialCase;
```

这使人联想到 Press 的 **Unification** 模式，它将 hook 函数和模板函数集成到单个的类 TH（template hook）中[Pree1995]，第 4.3.2 小节）。我们可以使用这个相同的类作为更通用的 **Template Method** 的基础（参见第 9.2.3 小节）。

9.2.7 Singleton 模式

Singleton（图 9.4，[Gamma1995]，第 127 页）是强制类与类的实例之间意义对应的一种模式。GOF 一书认为："类的惟一实例"是可以相互区别的惟一项，Gamma 模式特征描述中的差异字段是一个薄弱环节。Singleton 的例化中存在简单的差异性（0 或 1），其状态可以不同。其他内容均为常量。

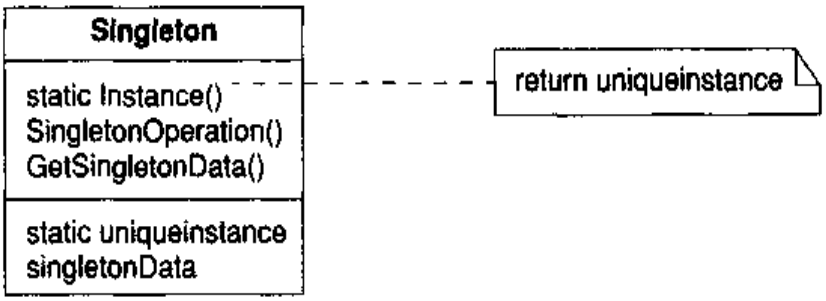


图 9.4 **Singleton** 的结构（改编自[Gamma1995]，第 127 页）

9.3 消极差异性的模式

表 9.1 在一些共同性背景下捕获了积极差异性的模式。与我们开发捕获消极差异性的语言特性的类似分类法一样，我们可以开发一个消极差异性的模式分类表。表 9.2 捕获了两种相关的情况，其中的模式很好地表达了消极差异性。

表 9.2 处理共同性和消极差异性的模式

共同性种类	差异性种类	绑　定	例　化	模　式
某些结构和算法	函数名或语义	编译或运行时间	可选	Adapter（第 9.3.2 小节）
相关运算，但非结构	类成员的取消	任意	是	Bridge 或 Envelope/Letter（第 9.3.1 小节）

注意：表 9.1 的格式类似于 C++方案领域差异性表（表 6.2）。绑定、例化和共同性类别对两个表都很重要。有时我们可以用语言特性捕获结构，有时我们可以用一种模式来捕获此种结构。表 9.1 更自由地描述了差异性，因为共同性类别（或者单独是 C++语言的共同性类别）不够强大，不足以捕获语义。

9.3.1 Bridge 模式

我们使用 **Bridge** 来处理违反主要的共同性的实现差异，这与积极差异性的情况类似。实际上，最好将 **Bridge** 当作一种消极差异性模式，积极差异性只是它的一种特殊情况。积极差异性全部取消了共同的结构，然后添加新的结构。

Bridge 可用于解析出函数成员关系和数据成员关系。我们来考虑一个 Message 抽象族，它通常需要校验和，单字节的 ACK（命令正确应答）和 NAK（无应答）消息除外。我们可以使用 **Bridge** 将这些差异性归入到一个实现“辅助”类 MessageBody 中。这里，我们使用私有多重继承（mix-in）来解析出校验和函数，我们还可以将此函数放在类层次结构的其他层次中：

```
class CheckSummedMessageBody: public MessageBody {
    . . . .
};
```

下面是 **Bridge** 层次结果的实现侧：

```
class MessageBody {
public:
    virtual void dispatch() = 0;
};

class CheckSummable {      // mix-in class
public:
    long checksum(const unsigned char*);
};

class FilePacketBody: public MessageBody,
    private CheckSummable {
    // CheckSummable is a mix-in
public:
    . . . .
private:
    using CheckSummable::checksum;
};

class AckBody: public MessageBody {
public:
    . . . .
    // no checksum member
    void dispatch() { . . . . }
private:
    . . . .
};
```

在应用接口侧上存在一个并行的结构：

```
class Message {
public:
    Message(const unsigned char *);
    void send() {
        . . . .
        implementation->dispatch();
        . . . .
    }
    . . . .
protected:
    MessageBody *implementation;     // the bridge
};

class AckMessage: public Message {
```

```
public:
    AckMessage(const unsigned char *c): Message(c) {
        . . . .
        implementation = new AckBody. . .
    }
};

. . . .
```

9.3.2 Adapter 模式

设计者可以用 **Adapter** 来删除某个抽象的部分接口，同时不影响其实现。共同性包括全部的基类算法和数据结构，而差异性存在于接口中。这与 **Strategy** 相反。**Adapter** 在规模等级上也不同于 **Strategy**，因为前者适用于整个类，而后者仅适用于单个函数。

作为一个例子，我们来考虑对 `List` 和 `Set` 的一个著名问题的分析（[Coplien1992]，第 6.5 节）：

```
template <class T> class List {
public:
    List();
    bool has(const T&) const;
    T head() const;
    T tail() const;
    void insert(const T&);
    void sort();
    unsigned size() const;
    . . . .
};
template <class T> class Set {
public:
    Set(): rep(new List<T>) { }
    void insert(const T &t) {
        if (!has(t)) rep->insert(t);
    }
    bool has(const T &t) { return rep->has(t); }
    unsigned size() const { return rep->size(); }
    // no sort, because there's no sense of ordering
private:
    List<T> *rep;
};
```

这是适配还是遏制？两者具有相同的语义（至少对于一一映射的情况），所以关键在于设计者如何看待它。

9.3.3 其他模式

第9.2节和第9.3节中的小节说明了来自流行文化中的模式是如何与多范型设计的原则相符合的。但这不是一个很彻底的分析，甚至对Gamma等人的成果解释得也不够充分。而且，《Design Patterns》一书[Gamma1995]只是汇集不断增长的设计模式的开始。读者应当仔细探究其他的模式，并随着经验的增加，将它们添加到表9.1和表9.2中。

更重要的是，很多模式捕获了多范型设计表达能力以外的设计考虑事项。这些模式是强大的设计辅助工具，对于从编程语言而不是[Gamma1995]的设计模式中分离出来的专用设计问题尤其如此。

9.4 作为模式助手的多范型工具

本章主要描述了作为多范型设计的辅助工具的模式。反过来也一样，模式也可以从多范型设计原则中获益。Gamma等人在绘制设计模式差异性表[Gamma1995]时在这方面已经迈出了一步，但他们的分类方法无法满足多范型设计的更宽设计空间的需要。共同性/差异性可以允当模式的很好的组织原则，这样就补充了Buschmann、Meunier等人的分类法（[PloP1995]，第134~135页，第428~430页）。

我们来考虑**Template Method**、**State**和**Strategy**。**Template Method**和**Strategy**的不同在于它们的差异性粒度，但其他方面类似。**Strategy**的第一个实现与**State**类似，但在绑定时间上稍有不同。（**Strategy**通常是一次性绑定的模式，而**State**是可以更加动态绑定的模式。）从多范型设计的观点来看，由于绑定时间的鲜明对照，**Strategy**的两种实现之间存在的差异可能多于**Strategy**和**State**之间存在的差异。模式并不强调这种区别；但多范型设计使之变得比较明显。

多范型设计帮助设计者查看作为某个区段上的点的这些模式，而不是作为解决具体问题的设计空间中的具体点。共同性和差异性帮助引出很多模式下（并非全部）的设计原则。正如在编码设计经验的更大语境中模式帮助我们适应多范型设计一样，多范型设计也通过解释形成设计模式核心的共同性和差异性类别，从而帮助软件设计者确定设计方向。

9.5 小结

有些软件设计模式只是我们在多范型设计中所发现的设计技术的简称。模式可以提升设计对话的水平，将其从具体的共同性和差异性类别提升至常见的类“丛”（constellation）和其他语言特性。这些组合中有很多都支持消极差异性。但单独的多范型设计是无法处理已经分类的整个模式区段的。

参考文献

[Barton+1994] Barton, John, and Lee Nackman. *Scientific and Engineering* C++. Reading, MA: Addison-Wesley. 1994.

[Beck1993] Beck, Kent. "Think Like an Object." *UNIX Review* 9, 10. September 1993. pp. 39–4.

[Bentley1988] Bentley, J. L. *More Programming Pearls: Confessions of a Coder.* Reading, MA: Addison-Wesley. 1988.

[Booch1994] Booch, Grady. *Object-Oriented Design with Applications, 2d ed.* Redwood City, CA: Benjamin/Cummings. 1994.

[Budd1995] Budd, Timothy. *Multi-Paradigm Programming in Leda.* Reading, MA: Addison-Wesley. 1995.

[Cain+1996] Cain, Brendan, James O. Coplien, and Neil Harrison. "Social patterns in productive software development organizations." *Annals of Software Engineering*, 2. December 1996. pp. 259–286. See http://www.baltzer.nl/ansoft/articles/2/ase004.pdf.

[Campbell+1990] Campbell, G. H., S. R. Faulk, and D. M. Weiss. "Introduction to Synthesis." Software Productivity Consortium, INTRO_SYNTHESIS-90019-N, Version 01.00.01. June 1990.

[Cockburn1998] Cockburn, Alistair. *Surviving Object-Oriented Projects: A Manager's Guide.* Reading, MA: Addison Wesley Longman. 1998.

[Constantine1995] Constantine, Larry. "Objects in your Face." In *Constantine on Peopleware.* Englewood Cliffs, NJ: Prentice-Hall. 1995. pp. 185–190.

[Coplien1992] Coplien, J. O. *Advanced C++ Programming Styles and Idioms*. Reading, MA: Addison-Wesley. 1992.

[Coplien1995a] Coplien, J. O. "A Generative Development-Process Pattern Language." *Pattern Languages of Program Design*. J. O. Coplien and D. Schmidt, eds. Reading, MA: Addison-Wesley, 1995. pp. 183-238.

[Coplien1996a] Coplien, J. O. "A Professional Dilemma." C++ *Report* 8, 3. New York: SIGS Publications. March 1996. pp. 80–89.

[Date1986] Date, C. J. *Introduction to Database Systems, 4th ed*. Reading, MA: Addison-Wesley. 1986.

[DeChampeaux+1993] DeChampeaux, Dennis, Doug Lea, and Penelope Faure. *Object-Oriented System Development*. Reading, MA: Addison-Wesley. 1993.

[Dijkstra1968] Dijkstra, E. W. "Notes on Structured Programming." *Structured Programming*. O. J. Dahl, W. Dijkstra, C. A. R. Hoare, eds. London: Academic Press. 1968.

[Flanagan1997] Flanagan, David. *Java in a Nutshell*. Bonn, Germany: O'Reilly and Associates. 1997.

[Fowler+1997] Fowler, Martin, and Scott Kendall. *UML Distilled: Applying the Standard Object Modeling Language*. Reading, MA: Addison Wesley Longman. 1997.

[Fusion1993] Coleman, Derek, et al. *Object-Oriented Development: The Fusion Method*. Englewood Cliffs, NJ: Prentice-Hall. 1993.

[Gamma1995] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley. 1995.

[GoldbergRubin1995] Goldberg, Adele, and Kenneth S. Rubin. *Succeeding with Objects: Decision Frameworks for Project Management*. Reading, MA: Addison-Wesley. 1995.

[Jerrell1989] Jerrell, Max E. "Function Minimization and Automatic Differentiation Using C++." *Proceedings of OOPSLA '89, SIGPLAN Notices*, 24, 10. October 1989. pp. 169–173.

[Jacobson+1992] Jacobson, Ivar, et al. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA: Addison-Wesley. 1992.

[Kiczales1994] Kiczales, Gregor. Keynote address at the 1994 ACM Conference on Object-Oriented Programs, Systems, Languages, and Applications. Portland, OR. Oct. 23–27, 1994.

[Kiczales1997] Kiczales, Gregor. "Aspect-Oriented Programming." *Computing Surveys* 28(4es). 1996. p. 154.

[Kuhn1970] Kuhn, Thomas. *Structure of Scientific Revolutions*. Chicago: University of Chicago Press. 1970.

[Liskov1988] Liskov, Barbara. "Data Abstraction and Hierarchy." *SIGPLAN Notices* 23, 5. May 1988.

[McConnell1997] McConnell, Steve M. *Software Project Survival Guide: How to be Sure Your First Important Project Isn't Your Last*. Microsoft Press. 1997.

[Meyer1992] Meyer, Bertrand. *Eiffel: The Language*. New York: Prentice-Hall. 1992.

[Meyers1992] Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Reading, MA: Addison-Wesley. 1992. Note: 2nd Edition published 1998.

[Neighbors1980] Neighbors, J. M. "Software Construction Using Components." Tech Report 160. Department of Information and Computer Sciences, University of California. Irvine, CA. 1980.

[Palsberg+1994] Palsberg, Jens, and Michael I. Schwartzbach. *Object-Oriented Type Systems*. Chichester, England: John Wiley & Sons. 1994.

[Parnas1976] Parnas, D. L. "On the Design and Development of Program Families." *IEEE Transactions on Software Engineering*, SE-2: March 1976. pp. 1–9.

[Parnas1978] Parnas, D. L. "Designing Software for Ease of Extension and Contraction." *Proc. 3rd Int. Conf. Soft. Eng.* Atlanta, GA. May 1978. pp. 264–277.

[PLoP1995] Coplien, J. O., and D. C. Schmidt, eds. *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley. 1995.

[PLoP1996] Vlissides, J., J. O. Coplien, and N. Kerth, eds. *Pattern Languages of Program Design — II*. Reading, MA: Addison-Wesley. 1996.

[PLoP1998] Martin, R., Dirk Riehle, and Frank Buschmann. *Pattern Languages of Program Design—3*. Reading, MA: Addison Wesley Longman. 1998.

[Pree1995] Pree, Wolfgang. *Design Patterns for Object-Oriented Software Development*. Reading, MA: Addison-Wesley. 1995.

[Reenskaug1996] Reenskaug, Trygve. *Working with Objects: The OOram Software Engineering Method*. Greenwich: Manning Publications. 1996.

[Rumbaugh1991] Rumbaugh, James. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall. 1991.

[Selic+1994] Selic, Bran, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. New York, NY: John Wiley & Sons, 1994.

[Shaw1994] Shaw, Mary. "Formalizing Architectural Connection." *Proceedings of the 16th International Conference on Software Engineering*. 1994.

[Snyder1986] Snyder, Alan. "Encapsulation and Inheritance in Object-Oriented Programming Languages." *SIGPLAN Notices* 21,11. November 1986.

[Stevens+1974] Stevens, Myers, Constantine. "Structured Design." *IBM Systems Journal*. 1974.

[Stroustrup1986] Stroustrup, B. *The C++ Programming Language*. Reading, MA: Addison Wesley Longman. 1997. p. 123. Note: 3rd Edition published in 1998.

[Tracz1995] Tracz, Will. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. Reading, MA: Addison-Wesley. 1995.

[Turner1993] Turner, Kenneth J. *Using Formal Description Techniques: An Introduction to Esterel, Lotos, and SDL*. Chichester, England: John Wiley & Sons. 1993.

[Ungar+1987] Ungar, David, and R. B. Smith. "Self: The Power of Simplicity." *Proceedings of OOPSLA 1987 (SIGPLAN Notices* 22(12)). New York: ACM Press, December 1987. pp. 227–241.

[Wegner1987] Wegner, Peter. "Dimensions of Object-Based Language Design." *SIGPLAN Notices* 22, 12. December 1987. pp. 168–182

[Weinberg1971] Weinberg, Gerald M. *Psychology of Computer Programming*. Van Nostrand. 1971.

[Weiss1999] Weiss, David J., and Chi Tau Robert Lai. *Family Based Domain Engineering*. Reading, MA: Addison Wesley Longman. 1999.

[Whorf1986] Whorf, B. J. Cited in *The C++ Programming Language*, by Bjarne Stroustrup. Reading, MA: Addison-Wesley. 1986. p. iii.

[Winograd1987] Winnograd, Terry. *Understanding Computers and Cognition: A New Foundation for Design*. Reading, MA: Addison-Wesley. 1987.

[Wirfs-Brock1990] Wirfs-Brock, Rebecca, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice-Hall. 1990.

[Yourdon1979] Yourdon, Ed. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Englewood Cliffs, NJ: Prentice-Hall. 1979.

[Snyder1986] Snyder, Alan. "Encapsulation and Inheritance in Object-Oriented Programming Languages." *SIGPLAN Notices* 21,11. November 1986.

[Stevens+1974] Stevens, Myers, Constantine. "Structured Design." *IBM Systems Journal*. 1974.

[Stroustrup1986] Stroustrup, B. *The C++ Programming Language*. Reading, MA: Addison Wesley Longman. 1997. p. 123. Note: 3rd Edition published in 1998.

[Tracz1995] Tracz, Will. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. Reading, MA: Addison-Wesley. 1995.

[Turner1993] Turner, Kenneth J. *Using Formal Description Techniques: An Introduction to Esterel, Lotos, and SDL*. Chichester, England: John Wiley & Sons. 1993.

[Ungar+1987] Ungar, David, and R. B. Smith. "Self: The Power of Simplicity." *Proceedings of OOPSLA 1987 (SIGPLAN Notices* 22(12)). New York: ACM Press, December 1987. pp. 227–241.

[Wegner1987] Wegner, Peter. "Dimensions of Object-Based Language Design." *SIGPLAN Notices* 22, 12. December 1987. pp. 168–182

[Weinberg1971] Weinberg, Gerald M. *Psychology of Computer Programming*. Van Nostrand. 1971.

[Weiss1999] Weiss, David J., and Chi Tau Robert Lai. *Family Based Domain Engineering*. Reading, MA: Addison Wesley Longman. 1999.

[Whorf1986] Whorf, B. J. Cited in *The C++ Programming Language*, by Bjarne Stroustrup. Reading, MA: Addison-Wesley. 1986. p. iii.

[Winograd1987] Winnograd, Terry. *Understanding Computers and Cognition: A New Foundation for Design*. Reading, MA: Addison-Wesley. 1987.

[Wirfs-Brock1990] Wirfs-Brock, Rebecca, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice-Hall. 1990.

[Yourdon1979] Yourdon, Ed. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Englewood Cliffs, NJ: Prentice-Hall. 1979.